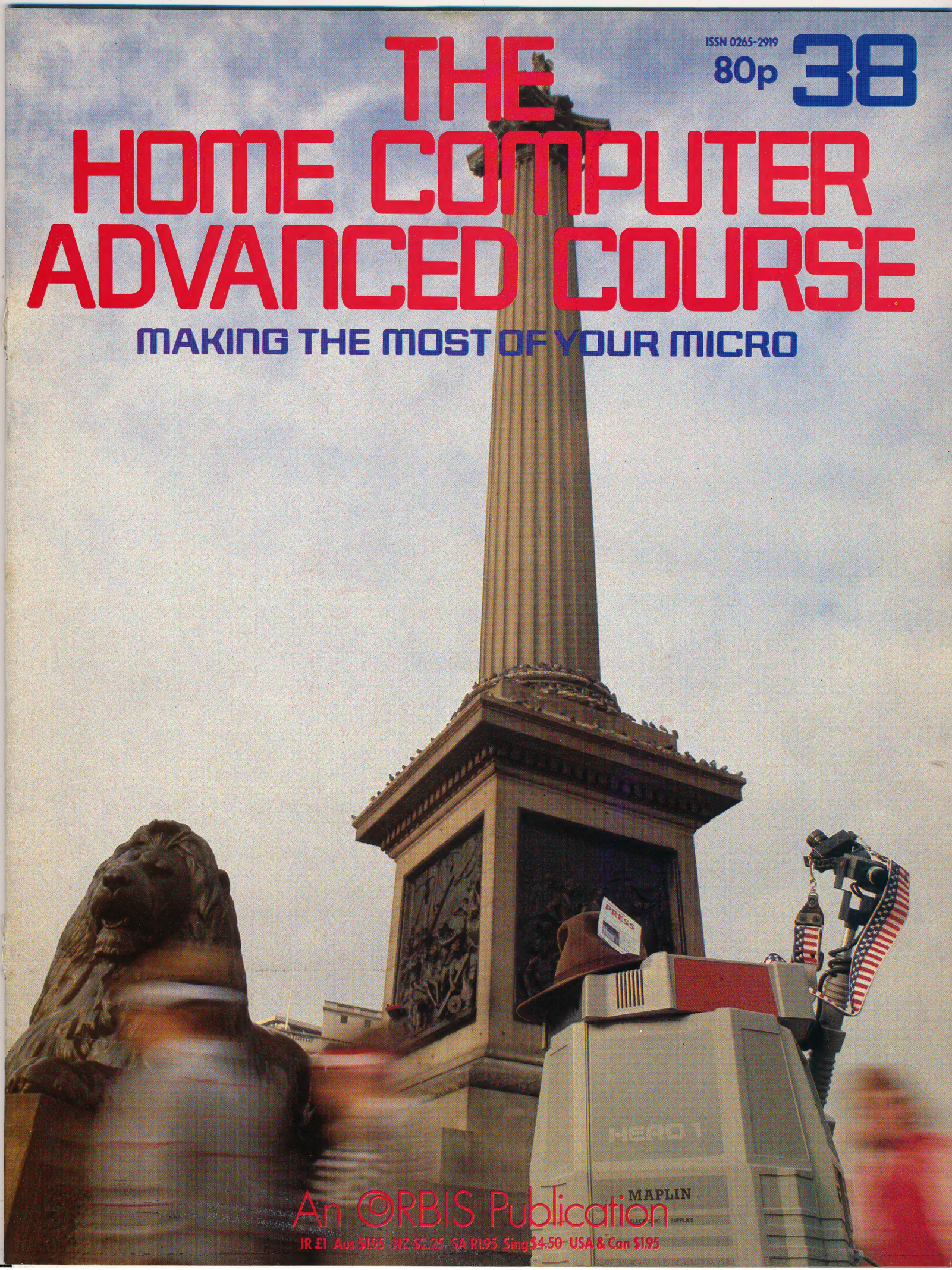
ISSN 0265-2919

80p 38

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



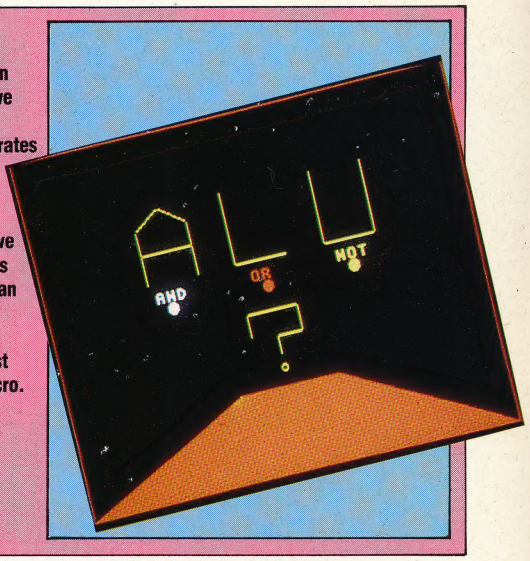
An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

- **Continuing our series on spreadsheet modelling, we look at Multiplan from Microsoft, which incorporates many new features.**
- **We begin a new programming project as we investigate the techniques involved in programming an adventure game.**
- **The Beasty is a low-cost robot arm for the BBC Micro.**



QUIZ

1) What method of speech synthesis is used in Flyerfox?
2) Apart from the BBC Micro and disk drive, what else is needed to use Graph Plan?
3) What is the 'grey scale'?
4) A sine wave is a pure curve. How can we change our analogue wave simulation to more accurately describe a sine wave?

Answers To Last Week's Quiz

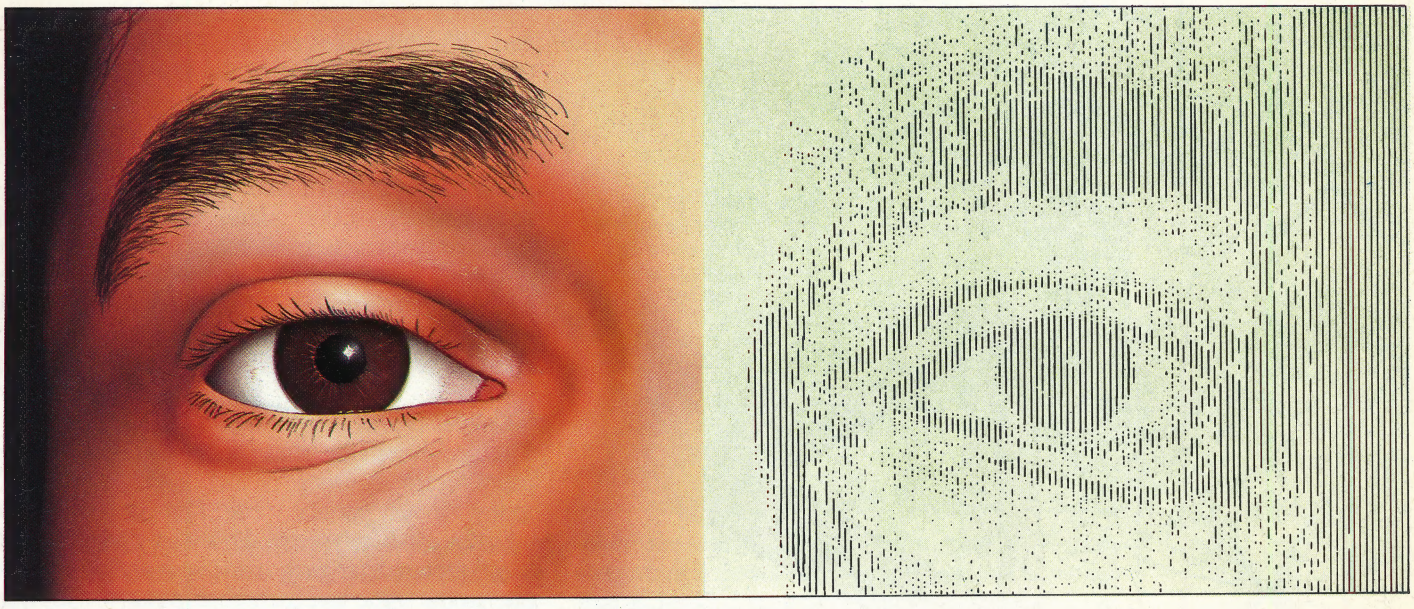
1) Stirling Mouse is a maze-following robot built by Nick Smith to compete in the British Micromouse contest. It was the first such mouse to solve the maze in competition.
2) Tape streamers are high-speed high-capacity tape recorders for storing data often used to back-up the contents of disks.
3) A saw-tooth wave is a common waveform in electronic sound. It produces a slightly harsher sound than the pure-tone sine wave output from most oscillators.

Editor Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Stephen Malone; **Sub Editor** Steve Mann; **Researcher** Melanie Davis; **Consultant Editor** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellar, Mike Curtis, Steve Colwill, Chris Naylor, Tony Harrington, Steve Malone, Ted Ball; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; © **APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984; Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Price: 80p/IR£1. Subscription: 6 months: £23.92. 1 Year: £47.84. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 80p. Subscription: 6 months air: £37.96. Surface: £31.46. 1 year air: £75.92. Surface: £62.92. Binder: £5.00. Airmail: £5.00. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 80p. Subscription: 6 months air: £43.94. Surface: £31.46. 1 year air: £87.88. Surface: £62.92. Binder: £5.00. Airmail: £8.31. **AMERICAS/ASIA/AFRICA** – Price: US/CANS1.95/80p. Subscription: 6 months air: £51.74. Surface: £31.46. 1 year air: £103.48. Surface: £62.92. Binder: £5.00. Airmail: £9.44. **SOUTH AFRICA** – Price: SA R1.95. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 80p. Subscription: 6 months air: £55.38. Surface: £31.46. 1 year air £110.76. Surface: £62.92. Binder: £5.00. Airmail: £9.84. **AUSTRALIA** – Price: AusS1.95. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZS2.25. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 6711. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in sterling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY A. PIDGEON

EYE ROBOT



STEVE CROSS

Robot Vs. Human Vision
The nature of the human's ability to see relies very heavily on the interaction between a complex system of nerves and receptors, all processed by the brain. Although a visual image consists of patterns of light and dark imprinted on the retina, the actual 'seeing' is done in the brain. A robot's brain also processes an image of light and dark patterns, but suffers from a much lesser degree of precision

Our investigation into the topic of robotics has now considered several different ways in which a robot may be programmed to act 'intelligently'. Here we examine the problems involved in enabling a robot to 'see' objects in the world around it.

Of all the human senses, vision is perhaps the most important. So central are visual perceptions to our understanding of the world that the phrase 'it is like trying to describe colour to a blind man' is often used to illustrate the difficulties involved in explaining a person's lack of understanding of a particular subject. Without vision, our knowledge of the world is severely restricted — and, similarly, a robot with no visual apparatus is equally handicapped. We have already seen how robots can use sensors to detect the presence of an object in their path; now we want to develop a system that will provide them with visual equipment that is as efficient as a human being.

The human eye has an iris that acts as a lens, controlling the amount of light entering it, and a retina onto which the lens focuses the image. But the fact is that the eye does not really 'see' anything at all; it is merely a transducer that converts one signal into another, more acceptable, form. The real job of seeing is carried out by the brain on the basis of the signals it receives from its sensors.

So in robot vision we can divide the subject into two quite distinct parts. The first concerns the construction of an appropriate 'eye' to act as a sensor for the robot's vision system; the second part consists of the computer processing that must be carried out before the robot can make sense of the signals from this sensor.

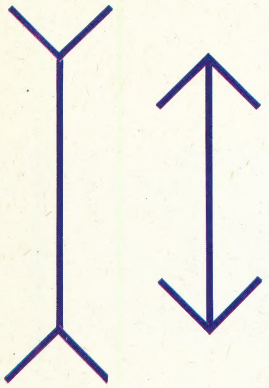
Constructing a robot eye is not too difficult. At its very simplest level, a photoelectric cell can act as a type of eye. This can give a signal that corresponds to the overall illumination of the field of view — as we have already seen, this can be useful if we simply want our robot to 'home in' on a bright light or to follow a white line painted on a dark background. The program to use this sensory input may also be simple because the information received is limited and there is a proportionately limited number of actions that the robot could follow as a result of such simple signals.

But we can hardly call this 'vision', in the sense that we understand the word. Specifically, we require a visual system that can build up a complete two-dimensional image of the world, enabling the robot's 'brain' to examine exactly the same information that is processed by the human brain and arrive at an answer.

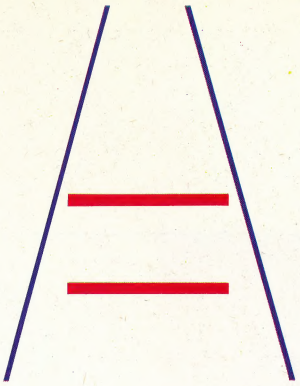
One answer to this problem uses a single photoelectric cell with a lens placed in front of it. This scans the image area in front of the robot, mechanically sweeping the entire field of vision until a complete picture has been built up; the picture can then be stored in the computer's memory. In practice, unfortunately, this method is slow and unreliable.

In most cases, however, the robot eye consists of a video camera of some sort. This camera may be the standard type that is used in television broadcasting, or it may be a specialised device that is designed specifically for robotic vision. Some forms of the latter type use special chips called Optical RAMs — these consist of RAM memory in which the value of each byte is set automatically

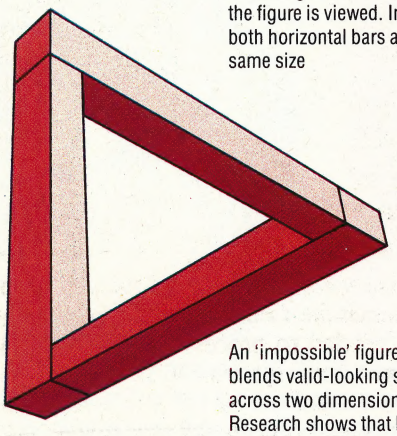
What You Think You See



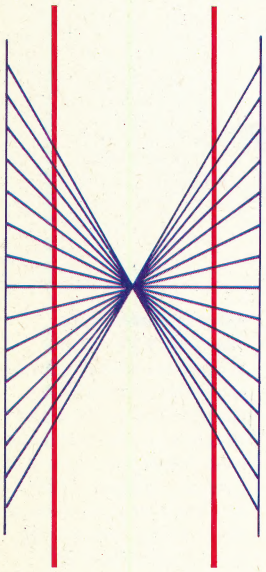
Which is longer? The Muller-Lyer illusion makes you think the vertical line in the left-hand figure is longer than the vertical line on the right. In fact, they are the same length



In the railway illusion, the upper horizontal bar appears to be longer, no matter how the figure is viewed. In fact, both horizontal bars are the same size

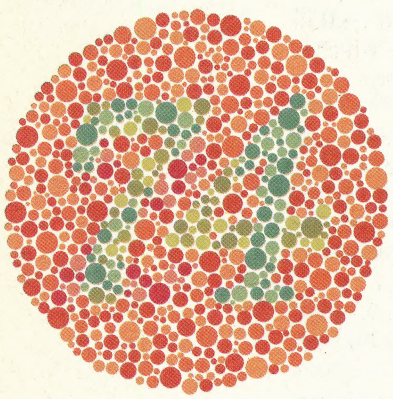


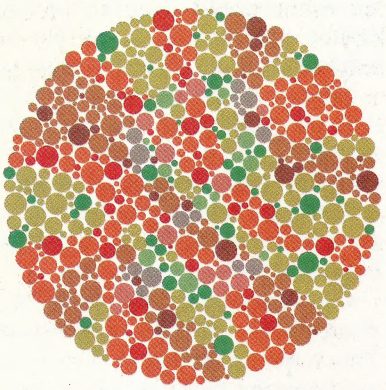
An 'impossible' figure, which blends valid-looking shapes across two dimensions. Research shows that humans are unable to recognise this as an object



KEVIN JONES

Although the vertical lines appear to bend, they are actually straight. The eye bends the image to conform with the spread of the rays





The colour patterns shown, taken from Ishihara's tests for colour blindness, measure red-green deficiencies. In the first example, persons with normal vision will see the number 74, while persons with a red-green deficiency will see the number 21. In the second example, normal vision will see nothing, or only a dim outline of a figure 2, while red-green deficient viewers will clearly see a 2

by the amount of light falling on that particular byte. These devices are becoming increasingly cheap and provide an area of RAM memory that contains all the information concerning the scene viewed by the robot's eye.

In general, the output from a robot eye is held in a two-dimensional array, each element of which contains a value that corresponds to the brightness of the light falling on that particular part of the scene being viewed. The number of elements in the array gives the *resolution* of the image, and the range of numbers that can be held in each array element determines the number of *grey scale* levels that can be discerned. Traditionally, in vision systems each element of the array is called a pixel, or 'picture element'. So an image array of 500 by 250 pixels that represents brightness levels by allocating one byte to each pixel would have a horizontal resolution of 500 pixels, a vertical resolution of 250 pixels, a total of 125,000 pixels, and 256 grey scales ranging from black to pure white. To give an idea of the detail that such a picture might provide, consider a standard television picture. The British system uses 625 vertical lines, so the vertical resolution is 625 pixels. To get a similar resolution across the screen, approximately 1,000 pixels would be needed (because the screen is wider than it is high), and the grey scale levels could be represented by the same single byte to give 256 brightness levels. A robot system with an equivalent resolution would give an acceptable image for the computer to process.

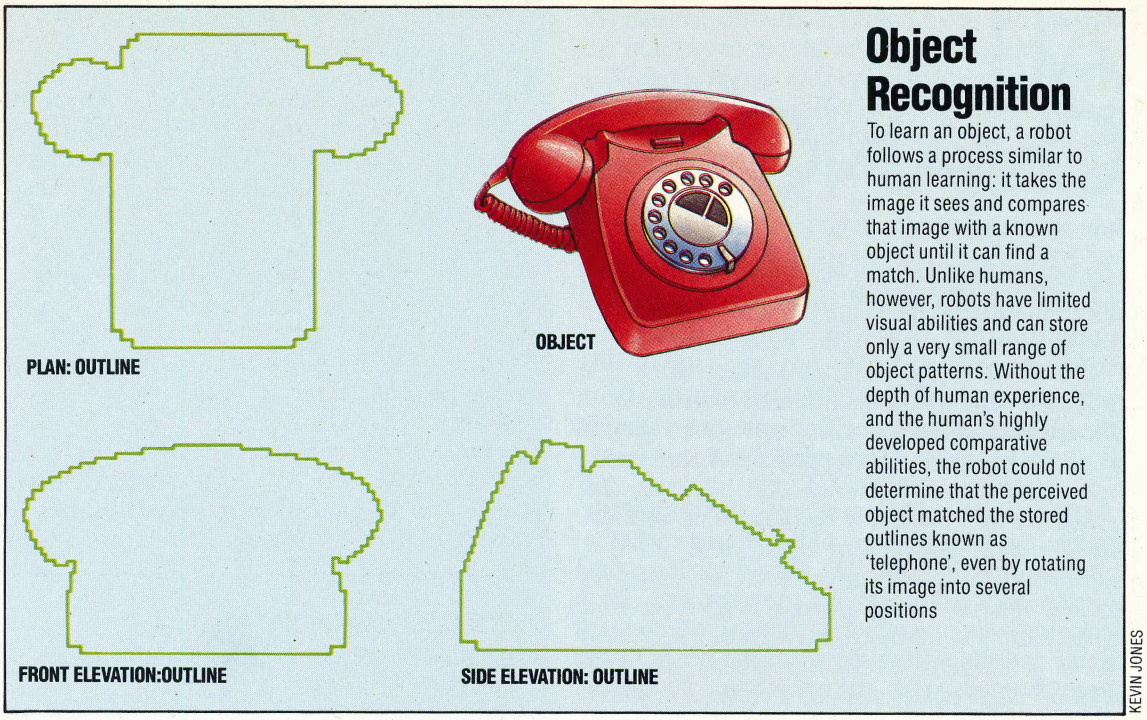
SCANNING THE IMAGE

The processes that must be undertaken by the robot 'brain' in order for this image to be 'seen' follow a set pattern. The first step involves adjusting the grey scale levels so that adjacent pixels with similar grey scale levels are 'smoothed out' to the same level. The computer works over the entire image area averaging levels so that any small irregularities are removed. Once this is done, the computer examines the image again, noting any adjacent pixels that have markedly different grey scale levels; these differences are then emphasised. The idea behind this is that important features in the image are probably marked by boundaries, such as lines and edges, which will show up as sudden changes in the grey scale level: the computer takes note of these, emphasising them to make sure that they stand out.

Once this has been done, the computer scans the image again, searching for all the really large changes in grey scale levels. It then uses these in much the same way as a human would join the dots in a puzzle to make up a picture. In most of these puzzles, the human is aided by the fact that the dots are numbered; the computer has no such help and must simply follow what seems to be a likely route. At the end of this process the robot will have an internal picture of the scene before it in which it has 'smoothed' the image out and drawn lines around those objects that seem to be important.

But is this 'seeing'? In fact, all the robot has done is to carry out certain transformations of the scene — in other words it still does not 'know' what it is looking at.

There are two solutions to this problem. The first is to program the robot with a set of rules that are expressed as a set of simple statements about the visual world. This is known as a 'bottom-up' approach to visual perception — so called because the robot starts with very simple things and tries to work out from them what it is seeing at a more complex level of understanding. The second approach is to program the robot with a set of



Object Recognition

To learn an object, a robot follows a process similar to human learning: it takes the image it sees and compares that image with a known object until it can find a match. Unlike humans, however, robots have limited visual abilities and can store only a very small range of object patterns. Without the depth of human experience, and the human's highly developed comparative abilities, the robot could not determine that the perceived object matched the stored outlines known as 'telephone', even by rotating its image into several positions

KEVIN JONES

objects that it is likely to see and then have it examine the image to see if any of these objects are present. This is known as the 'top-down' approach, so called because the robot begins with a very complex, high-level idea about what it might be seeing and then checks to see if its actual visual input corresponds.

OBJECT RECOGNITION

To illustrate the difference between the two methods, consider a robot that is looking at a table. The bottom-up approach consists of analysing the image and finding that it contains four vertical parts and, near the top of them, a large horizontal surface. This corresponds to the pre-programmed knowledge that a large surface could be resting on four legs and that this structure is called a table. The top-down approach would start with the robot looking at the table and asking itself 'is that a table?'. It can ask itself this question because it has an internal model of a table against which it checks its visual input.

In general, the bottom-up approach enables the robot to see things that it has never encountered, and to understand something about them — though to do this it requires a great deal of detailed programming to give it the necessary basic rules about the world that it will encounter. However, the top-down approach allows the robot to recognise only objects that it already has some internal knowledge of — so anything new to it will cause problems.

Both methods are used by robot designers, and sometimes a mixture of the two methods is utilised. It seems likely that humans use similar methods in their own visual perception, but we do it automatically and are unaware of these processes taking place.

But robot vision is, as yet, far from perfect. There are several reasons for this. One of the most important is the sheer amount of processing power that is needed to process an image. Remember that our example system had 125,000 pixels stored as one byte each — over 122 Kbytes of memory must thus be processed for each image. Although we have simplified our description, many of the processes that must be carried out on each pixel are mathematically fairly complicated. If the robot has to observe the world about it in 'real time' (i.e. it considers events as they happen), then 25 different images will be received every second (this is also true of television cameras). This means that the robot would need to process over 3,050 Kbytes of data each and every second — which is roughly equivalent to the contents of more than one dozen floppy disks!

To deal with the problem of processing, two approaches may be considered. One is to develop special-purpose hardware to perform the image-processing (such hardware is now becoming available). Alternatively, the image resolution and number of grey scale levels may be reduced to match existing hardware. This will result in the image being processed more quickly, but the picture quality will be poorer.

At present however, the subject of robot vision is still not completely understood — any more than the details of human sight are. Robots often make mistakes when using a vision system. It may well be that, ultimately, the only answer will be to develop systems in which the robot 'learns' to see things instead of being programmed in detail about what it can and cannot see. And it may eventually prove to be the case that a robot can never 'see' things properly until a way has been developed to give it a much greater knowledge of the surrounding world — knowledge that is comparable to our own, in fact.

ALL CHANGE

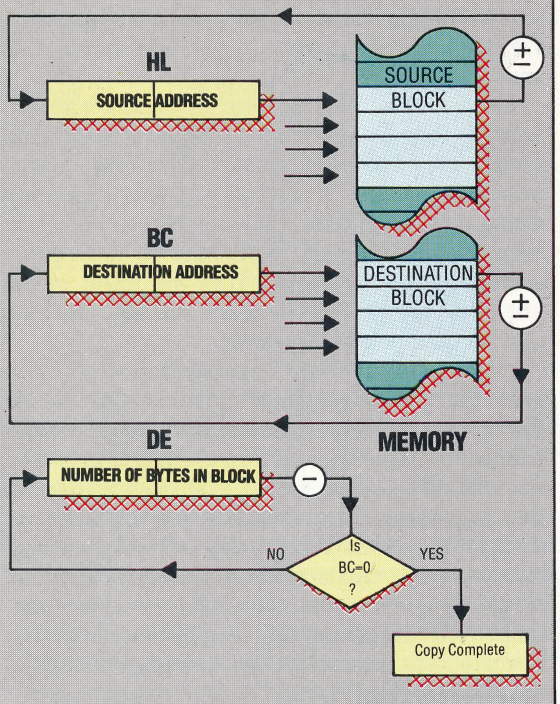
To implement the variable replace program on the Spectrum, we must use a different method from the BBC and Commodore versions. Instead of having the utility program in a different area of memory from the program it is working on, the variable replace program is merged onto the end.

As the variable replace program scans through the program it makes a copy of the altered version in an area above RAMTOP. The altered version is then copied back to the main program area by a machine code program that adjusts the amount of space available if the length of the program has been changed, so that the new version fits into the BASIC text area.

The first part of the BASIC program is similar to the variable search program (see page 665). There are some extra variables, including Altprog, which points to the start of the area reserved for the copy of the program, and Altpointer, which keeps track of where the next byte in the altered program should go. The main changes involve copying the program, instead of just reading it. The copying is done by the subroutine at line 9800, which copies

Automatic Writing

The Z80 LDIR and LDDR op-codes are block transfer instructions using automatic increment or decrement: the HL register is initialised to point to the start of the source block, DE must point to the start of the destination, and BT must contain the number of bytes in the block. LDIR and LDDR then copy the source byte to the destination byte, automatically increment or decrement HL and DE, and decrement BC until it reaches zero, when the copy is deemed complete. Notice that LDIR is a 'dumb' copy (see page 726) — the programmer's intelligence is assumed



KEVIN JONES

Variable Replace Program

```
9000 INPUT "Name to search for? "; LINE
t$
9005 INPUT "Replace by?"; LINE r$
9010 FOR i=1 TO LEN (t$)
9020 IF t$(i)>="a" AND t$(i)<="z" THEN
LET t$(i)=CHR$ (CODE (t$(i))-32)
9030 NEXT i
9040 LET TokenforREM=234
9050 LET Quote=34
9060 LET Newline=13
9070 LET Underscore=95
9080 LET Number=14
9090 LET PROG=23635
9100 LET Textpointer=PEEK (PROG)+256*PEE
K (PROG+1)
9102 LET Altprog=46000
9105 LET Altpointer=Altprog
9110 LET Lineno=256*PEEK (Textpointer)+P
EEK (Textpointer+1)
9111 PRINT lineno
9120 IF Lineno>=9000 THEN  GO TO 9600
9130 LET q=2: GO SUB 9800
9135 LET Lengthaddr=Altpointer
9140 LET Nextline=Textpointer+2+PEEK (Te
xtpointer)+256*PEEK (Textpointer+1)
9150 LET q=2: GO SUB 9800
9160 LET Byte=PEEK (Textpointer): LET q=
1: GO SUB 9800
9170 IF Byte=Newline THEN  GO TO 9110
9180 IF Byte<>TokenforREM THEN  GO TO 92
20
9190 REM Copy REM  unaltered
9200 LET q=Nextline-Textpointer: GO SUB
9800
9210 GO TO 9110
9220 IF Byte<>Quote THEN  GO TO 9280
9230 REM Copy anything between quotes, b
ut stop at end of line in case of unmatc
hed quote
9235 LET q=1
9240 IF PEEK (Textpointer+q-1)=Newline T
HEN  GO SUB 9800: GO TO 9110
9250 IF PEEK (Textpointer+q-1)=Quote THE
N  GO SUB 9800: GO TO 9160
9260 LET q=q+1
9270 GO TO 9240
9280 REM Copy 5-byte binary number
9290 IF Byte=Number THEN  LET q=5: GO SU
B 9800: GO TO 9160
9310 REM First character of name must be
 upper or lower case letter
9320 IF Byte>=CODE ("A") AND Byte<=CODE
("Z") THEN  LET c$=CHR$ (Byte): GO TO 93
70
9330 REM Use upper case instead of lower
 case
9340 IF Byte>=CODE ("a") AND Byte<=CODE
("z") THEN  LET c$=CHR$ (Byte-32): GO TO
 9370
9360 GO TO 9160
9370 LET n$=""
9380 LET n$=n$+c$
9400 REM Letter, digit or underscore aft
er first character of name
9410 IF PEEK (Textpointer)>=CODE ("A") A
ND PEEK (Textpointer)<=CODE ("Z") THEN
LET c$=CHR$ (PEEK (Textpointer)): LET Te
xtpointer=Textpointer+1: GO TO 9380
9420 REM Use upper case instead of lower
 case
9430 IF PEEK (Textpointer)>=CODE ("a") A
ND PEEK (Textpointer)<=CODE ("z") THEN
LET c$=CHR$ (PEEK (Textpointer)-32): LET
 Textpointer=Textpointer+1: GO TO 9380
9440 IF PEEK (Textpointer)>=CODE ("0") A
ND PEEK (Textpointer)<=CODE ("9") THEN
LET c$=CHR$ (PEEK (Textpointer)): LET Te
xtpointer=Textpointer+1: GO TO 9380
9450 IF PEEK (Textpointer)=Underscore TH
EN  LET c$=CHR$ (PEEK (Textpointer)): LE
T Textpointer=Textpointer+1: GO TO 9380
9460 REM End with $ for string variable
9470 IF PEEK (Textpointer)=CODE ("$") TH
EN  LET n$=n$+"$": LET Textpointer=Textp
ointer+1: GO TO 9500
9480 REM ( if array or function
9490 IF PEEK (Textpointer)=CODE ("(") TH
EN  LET n$=n$+CHR$ (PEEK (Textpointer)):
 LET Textpointer=Textpointer+1
```

```
9500 IF n$=t$ THEN  LET n$=r$
9505 LET Altpointer=Altpointer-1
9510 FOR p=1 TO LEN (n$)
9520 POKE Altpointer,CODE (n$(p))
9530 LET Altpointer=Altpointer+1
9540 NEXT p
9550 IF n$<>r$ THEN  GO TO 9160
9560 LET LengthLow=PEEK (Lengthaddr)+LEN
 (r$)-LEN (t$)
9570 IF LengthLow>255 THEN  LET Lengthlo
w=LengthLow-256: POKE Lengthaddr+1,1+PEE
K (Lengthaddr+1)
9580 POKE Lengthaddr,LengthLow
9590 GO TO 9160
9599 REM Prepare to move altered program
 back to main program area
9600 LET Oldlength=Textpointer-(PEEK (PR
OG)+256*PEEK (PROG+1))
9610 LET Newlength=Altpointer-Altprog
9620 POKE 45060,Newlength-256*INT (Newle
ngth/256)
9630 POKE 45061,INT (Newlength/256)
9660 POKE 45056,Textpointer-256*INT (Tex
tpointer/256)
9670 POKE 45057,INT (Textpointer/256)
9680 IF Oldlength=Newlength THEN  RANDOM
IZE USR (45084)
9690 IF Oldlength<Newlength THEN  LET X=
Newlength-Oldlength
9700 IF Oldlength>Newlength THEN  LET X=
Textpointer-(Oldlength-Newlength)
9710 POKE 45058,X-256*INT (X/256)
9720 POKE 45059,INT (X/256)
9730 IF Oldlength<Newlength THEN  RANDOM
IZE USR 45062
9740 IF Oldlength>Newlength THEN  RANDOM
IZE USR 45074
9800 FOR p=Textpointer TO Textpointer+q-
1
9810 POKE Altpointer,PEEK (p)
9820 LET Altpointer=Altpointer+1
9830 NEXT p
9840 LET Textpointer=p
9850 RETURN
9900 SAVE "REPLACE" LINE 9910: SAVE "REP
MC"CODE 45064,37: STOP
9910 CLEAR 45055: LOAD "REPMC"CODE
```

Machine Code Loader

```
  10 CLEAR 45055
  20 LET a=45062
  30 FOR l=1000 TO 1040 STEP 10
  40 LET s=0
  50 FOR a=a TO a+7
  60 READ b
  70 POKE a,b
  80 LET s=s+b
  90 NEXT a
 100 READ c
 110 IF s<>c THEN  PRINT "DATA ERROR IN
LINE ";l: STOP
 120 NEXT l
1000 DATA 42,0,176,237,75,2,176,205,913
1010 DATA 85,22,24,10,42,0,176,237,596
1020 DATA 91,2,176,205,229,25,33,176,937
1030 DATA 179,237,91,83,92,237,75,4,998
1040 DATA 176,237,176,207,255,0,0,0,1051
```

Replace Assembly Program

```
0000             MKROOM  EQU     $1655
0000             RCLAM1  EQU     $19E5
0000             T0      EQU     $B000
0000             T1      EQU     $B002
0000             T2      EQU     $B004
0000             ALTPRG  EQU     $B3B0
0000             PROG    EQU     $5C53
B006                     ORG     $B006
B006  2A00B0     UP      LD      HL, (T0)
B009  ED4B02B0           LD      BC, (T1)
B00D  CD5516             CALL    MKROOM
B010  180A               JR      COPY
B012  2A00B0     DOWN    LD      HL, (T0)
B015  ED5B02B0           LD      DE, (T1)
B019  CDE519             CALL    RCLAM1
B01C  21B0B3     COPY    LD      HL, ALTPRG
B01F  ED5B535C           LD      DE, (PROG)
B023  ED4B04B0           LD      BC, (T2)
B027  EDB0               LDIR
B029  CF                 RST     8
B02A  FF                 DB      $FF
```

the number of bytes specified by q, and updates the pointers to the old and new programs.

Variable names are copied by the code starting at line 9500, and, if the variable name has been changed, the two bytes at the beginning of the line that hold the length of the line are altered to reflect the change in length.

After the whole program has been altered and copied to the new area, the BASIC program calculates the values needed by the machine code for copying back the altered program, and then POKEs these values into the memory locations where the machine code expects to find them.

If the new and old programs are the same length, the new program can be copied into the same space that is occupied by the old program. In this case, the only information needed by the machine code program is the program's length.

If the new program is longer than the old program, we have to make extra space in the program area by moving up the variable replace routine, which we want to keep. The extra space is made by calling the ROM subroutine, MAKE-ROOM, at address 1655 hex. When MAKE-ROOM is called, the HL register pair must contain the address after the place where space is to be made, and the BC register pair must hold the length of the space needed. The value required for HL is just the final value of the variable Textpointer, and the value for BC is the difference between the old and the new lengths.

If the new program is shorter than the old, we have to move the variable replace program down. We can do this by using the ROM subroutine RECLAIM-1 at address 19E5 hex. When RECLAIM-1 is called, the HL register pair must hold the address of the first byte to be left alone, and the DE register pair must contain the address of the first byte to be reclaimed. The value required for HL is again the final value of the variable Textpointer, and the value required for DE is calculated by subtracting the difference between the old and new lengths from Textpointer.

The altered program is copied back to the main program area by the block move instruction LDIR (LoaD with Increment and Repeat). The start address of the altered program area is loaded into HL, the start address of the main program area into DE, the length of the altered program into BC, and then the LDIR instruction moves the whole of the altered program, byte to byte.

The last two lines in the Assembly language program use another ROM routine, at address 8. This is the 'report' routine that prints an error message and other comments. The routine is called by the RST 8 instruction, and the report produced is specified by the byte following the RST 8 instruction. The value of the byte is one less than the report number, so FF hex, or -1, gives the OK or Program finished report; 0 gives NEXT without FOR, and so on. The machine code program ends with RST8, instead of the usual RET instruction, to avoid returning to the BASIC program that has been moved.

SETTING THE TONE

In the last instalment of Workshop we looked at the creation of digitally synthesised sound using a digital-to-analogue converter, and designed a machine code program to generate three types of waveform: square, saw-tooth and sine waves. Now we look at two other important sound parameters: volume and pitch.

The volume of a tone is determined by the range of oscillation of the waveform generating the tone. In other words, volume depends on the difference between the maximum value of the waveform and the minimum value. This property of a sound wave is called the *amplitude.*

Using a simple BASIC program to oscillate values placed in the user port register we can demonstrate how easily amplitude can be controlled in a digital waveform.

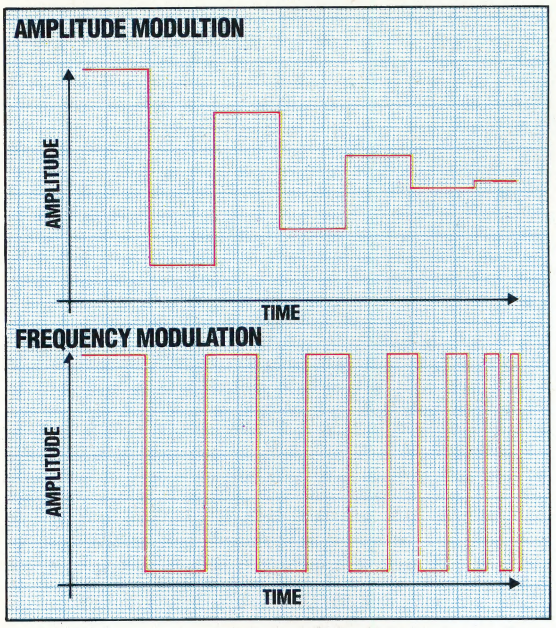
```
10 REM **** CBM BAS FREQ/AMPLITUDE ****
20 :
25 DDR=56579:DATREG=56577
30 POKE DDR,255:REM ALL OUTPUT
40 FOR I=255 TO 0 STEP-15
50 FOR J=1 TO 100
60 POKE DATREG,I:POKE DATREG,0
70 NEXT J,I

800 REM **** SAMPLE PROGRAM ****
805 :
806 UP$=CHR$(145)
810 DIV=49798: REM AMPLITUDE FACTOR LOCATION
820 DEL=49799: REM DELAY FACTOR LOCATION
830 TME=49800: REM DURATION FACTOR LOCATION
840 CALL=49801:REM PROGRAM START ADDRESS
850 :
860 DDR=56577:POKEDDR,255:REM ALL OUTPUT
870 :
880 PRINTCHR$(147):REM CLEAR SCREEN
890 PRINT:INPUT"AMPLITUDE FACTOR 0-7";AF
900 IF AF<0 OR AF>7 THEN PRINT UP$;UP$;:GOTO890
910 POKE DIV,AF
920 :
930 PRINT:INPUT"DELAY FACTOR 1-101";DF
940 IF DF<1 OR DF>101 THEN PRINT UP$;UP$;:GOTO930
945 POKE DEL,DF
950 :
960 PRINT:INPUT"DURATION FACTOR 0-15";TF
970 IF TF<0 OR TF>15 THEN PRINTUP$;UP$;:GOTO960
980 POKE TME,TF
990 :
1000 SYS CALL
1010 GETA$:IFA$="" THEN 1010
1020 IFA$="X" THEN 880:REM RESTART
1030 GOTO 1000REM ANOTHER BEEP
```

At the start of the program a crude square wave is generated that oscillates between 255 and 0. This means that the amplitude of the wave is 255. As the program runs, the upper value placed in the data register is reduced in steps of 15. As the upper value is decreased, so the amplitude decreases — and the effect of this, when monitoring the sound produced through a stereo amplifier or headphones, is that the volume of the tone gradually fades away to nothing. So the volume of a digitally synthesised tone can be controlled by limiting the range of values placed in the user port data register.

The pitch of a note is governed by the frequency of the generating wave; that is the number of waveform cycles per second — the larger the number of waveforms produced per unit time, the higher the pitch of the note heard.

Modular Construction
Waveforms can be modulated either by frequency or by amplitude. Frequency alters the pitch of the tone heard and is determined by the number of waveform cycles output per second. Amplitude alters the volume of the tone and is the difference between the maximum and minimum values in a cycle. The diagrams show how amplitude can be modulated so that the tone produced gradually decreases in volume and how frequency can be modulated to produce a tone which rises in pitch



Frequency can be controlled digitally in two main ways. The first is to increase frequency from a bottom limit by taking fewer samples of the waveform. If a wave were split into 100 samples, for example, then a machine code program would take a certain length of time to place each value in succession into the data register, and so a certain number of complete waveforms could be produced per second. The number of samples obviously governs the frequency of the tone heard. To double the frequency, the machine code program could, instead, take every second value only from the data table defining the wave. The frequency could be tripled by taking every third value, and so on. There are two drawbacks to this method. The first is that small frequency adjustments are difficult to make without distorting the shape of the wave. Secondly, as the frequency of the wave increases, the wave generated bears less and less relation to the original wave shape as fewer samples are used.

An alternative method is start with a loop that steps through the waveform data as fast as possible, thus providing a maximum frequency. Frequency can then be adjusted by inserting short delays in the loop. This gives us much more accurate control over the tone frequency, but means that the number of samples making up the waveform must be small if a reasonably high maximum frequency is to be obtained. We shall employ the second of these two methods to

produce a machine code program that will enable us to control both frequency and volume.

The best method of reducing the waveform amplitude, while retaining the overall shape of the wave, is to divide each value in the waveform table by a constant. This can be done in two ways: after each value is loaded into the accumulator but before the value is placed in the data register, or prior to entering the main program loop. The first method will increase the amount of time required to execute each cycle of the main loop, and as this factor limits the maximum frequency obtainable we should opt for the second method. A second table is produced from the original waveform table by dividing each value taken from the original table by a constant, then placing the result in the new table. The new table is then used for waveform data by the main program loop. The division method used is crude. An amplitude constant dictates the number of times that the table value is shifted right. As each shift right is an integer division by two, the effect of using an amplitude factor of 'n' is to divide each table by 2n.

A rather clever method is employed when the amplitude factor is zero. In this case we use the original table and the program modifies itself to specify the base address of the original table, rather than that of the table of divided values.

Execution of the main loop of the program can be delayed by inserting a small piece of code that does nothing except take time to execute. Normally this is done by decrementing either an eight-bit number in one of the index registers, or a 16-bit number in memory, from a set delay value to zero. We find that if we calculate the maximum delays produced by these two methods, decrementing an eight-bit number will provide sufficient delay.

The main problem is not providing enough delay — i.e. producing the lowest frequency — but providing the minimum delay; in other words, producing the maximum frequency. On page 732 we used a waveform divided into 80 steps. The extra code required for the delay slows down execution time so much that it is no longer practical to have this number of steps. The code making up the main loop is shown below.

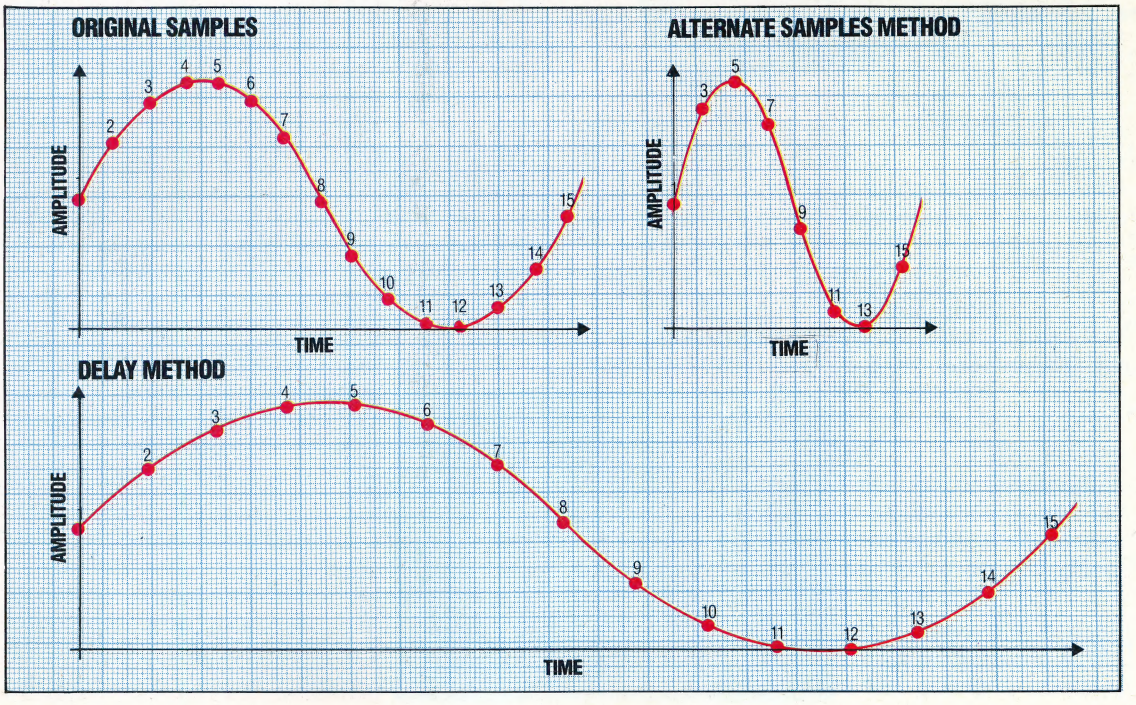
```
MAIN                                  Time in machine cycles
        LDX #$00                      2
NEXVAL
        LDA AMPTAB,X                  4
        LDY DELAY                     4
MORDEL
        DEY                           2
        BNE MORDEL                    3 (2 if loop fails)
        STA PORT                      4
        INX                           2
        CPX # STEPS                   2
        BNE NEXVAL                    3 (2 if loop fails)
```

The total number of machine cycles required is given by: $2+(4+4+(2+3)\times delay-1+4+2+2+3) \times steps -1 = 1+(18+5\times delay) \times steps$: and a minimum delay value of 1 yields the maximum frequency as follows: max freq = $1000000 \div (1+23\times steps)$.

For a maximum frequency of around 3,000 Hz, this formula gives the number of steps as 15. This is the number of samples of the waveform we should use to produce a reasonable maximum frequency. Using 15 steps, the original formula can be written as: no of machine cycles = $271+75\times delay$.

If we require a minimum frequency of, say, 128 Hz (about two octaves below middle C) then the delay value will be 101. This value can be held and decremented by an index register.

The final problem produced by an altered frequency is that, for a set number of repeats of the loop delay, given the duration of the tone produced will decrease as the frequency increases. This is because as the frequency increases, the main loop takes less time to execute. To balance this we must



LIZ DIXON

Stretching The Snake
The frequency of a digitally sampled wave can be altered by taking fewer samples or by inserting a delay between each sample value. If the original look-up table for the wave contains 15 samples of the wave, then the frequency of the output wave can be doubled by taking every other sample only. Alternatively all 15 values can be output inserting a delay to double the time taken to output the entire wave, halving the frequency. The first method allows many samples to be used at lower frequencies but affords only crude frequency control. The second method allows much finer frequency control but means that fewer samples must be used

include a calculation that will set the number of repeats required to produce a tone for a set duration, no matter what the frequency of the tones. If we make the unit of duration 1/50th of a second then the number of repeats for a given delay value is: 1000000÷(50×(271+75×delay). To calculate this in machine code would be difficult and slow so instead we can set up a look-up table from BASIC, giving a value from the above formula for each value of delay (i.e. from 1 to 109). The machine code routine then simply has to look up the repeat value and store it in a decrementer, to produce a tone lasting for 1/50th of a second.

Commodore owners with an assembler can type in the source code listing and assemble it to produce an object program that can be saved. The calling program will load back the object program from disk or cassette and set up the look-up tables. Type this in and run it. Type NEW and enter the sample BASIC program that gives the various location addresses used by the machine code program. RUN this program with the buffer box and D/A converter set up as shown on page 732. If you do not have an assembler, then type in and run this BASIC loader before running the calling program. If you use the BASIC loader then lines 45 and 50 of the calling program can be omitted.

BBC owners can simply type in the BBC version as written and RUN.

```
 10 REM BASIC LOADER FOR FREQ/AMP PROGRAM
 20 :
 30 FOR I=49801 TO 49911
 40 READ A:POKEI,A
 50 CC=CC+A
 60 NEXT I
 70 READ CS:IF CC<>CS THENPRINT
    "CHECKSUM ERROR":STOP
100 DATA172,134,194,208,8,169,0,141
110 DATA204,194,76,174,194,169,15,141
120 DATA204,194,162,15,172,134,194,189
130 DATA0,194,24,106,157,15,194,136
140 DATA208,248,202,16,239,120,169,0
150 DATA141,133,194,174,135,194,189,30
160 DATA194,172,136,194,240,7,10,46
170 DATA133,194,136,208,249,141,132
180 DATA194,162,0,189,15,194,172,135
190 DATA194,136,208,253,141,1,221,232
200 DATA224,15,208,239,173,132,194,56
210 DATA233,1,141,132,194,173,133,194
220 DATA233,0,141,133,194,208,218,169
230 DATA0,205,132,194,208,211,88,96
240 DATA16186:REM*CHECKSUM*
```

```
 10 REM **** CALLING PROGRAM ****
 20 REM ****        AND        ****
 30 REM ****  TABLE SET UP     ****
 40 :
 45 DN=8:REM IF CASSETTE DN=1
 50 IF A=0 THEN A=1:LOAD"FREQ.HEX",DN,1
 60 :
 70 REM **** SET UP SHAPE TABLE ****
 75 :
 80 S=15:TB=12*4096+2*256
 90 FOR I=0 TO S-1
100 Y=127*SIN(X)+127
110 POKE TB+I,Y
120 X=X+2/S
130 NEXT I
140 :
150 REM **** SET UP FREQ/DELAY TABLE ****
160 :
170 TB=TB+2*S
180 FOR D=0 TO 101
190 TV=10^6/(50*(271+75*D))
200 POKETB+D,TV
210 NEXT D
```

```
;++++++++++++++++++++
;++++++++++++++++++++
;++                  ++
;++     CBM 64       ++
;++    FREQUENCY     ++
;++ AND AMPLITUDE    ++
;++   GENERATION     ++
;++                  ++
;++++++++++++++++++++
;++++++++++++++++++++
;
STEPS  = 15       ;NO. OF STEPS PER WAVE CYCLE
PORT   = 56577
;
*=     $C200
SHAPTB *=*+STEPS          ;WAVE SHAPE TABLE
AMPTAB *=*+STEPS          ;AMPLITUDE TABLE
LOOPTB *=*+102            ;FREQ/DELAY TABLE
COUNT  *=*+2              ;LOOP COUNTER
DIVISN *=*+1              ;DIV OF WAVE FACTOR
DELAY  *=*+1              ;DELAY FACTOR
TIME   *=*+1              ;NOTE DURATION FACTOR
;
;++++ SET AMPLITUDE TABLE ++++
;
       LDY DIVISN
       BNE CONT
       LDA #<SHAPTB       ;MODIFY PROGRAM
       STA NEXVAL+1       ;TO LOAD SHAPTB
       JMP INITC
;
CONT
       LDA #<AMPTAB
       STA NEXVAL+1
       LDX #STEPS
NEXT
       LDY DIVISN
       LDA SHAPTB,X
MORE
       CLC
       ROR A
       STA AMPTAB,X
       DEY
       BNE MORE
       DEX
       BPL NEXT
;
;++++ SET COUNT VALUE ++++
;
INITC
       SEI
       LDA #$00
       STA COUNT+1        ;INIT COUNT HIBYTE
       LDX DELAY
       LDA LOOPTB,X
       LDY TIME
       BEQ NOMULT
MULT
       ASL A
       ROL COUNT+1
       DEY
       BNE MULT
NOMULT
       STA COUNT
;
;++++ MAIN PROGRAM LOOP ++++
;
MAIN
       LDX #$00
NEXVAL
       LDA AMPTAB,X
       LDY DELAY
MORDEL                    ;DELAY LOOP
       DEY
       BNE MORDEL
;
       STA PORT
       INX
       CPX #STEPS
       BNE NEXVAL
;
;++++ DECREMENT COUNT ++++
;
       LDA COUNT
       SEC
       SBC #$01
       STA COUNT
       LDA COUNT+1
       SBC #$00
       STA COUNT+1
       BNE MAIN
       LDA #$00
       CMP COUNT
       BNE MAIN
       CLI
       RTS
```

```
 15REM ****************************
 20REM **                        **
 25REM **          BBC           **
 30REM ** FREQUENCY & AMPLITUDE  **
 40REM **        GENERATION      **
 50REM **                        **
 60REM ****************************
 90MODE 7
 95steps=15:port=&FE60
 97ddr=&FE62:? ddr=255:REM ALL OUTPUT
100HIMEM=HIMEM-&101:REM RESERVE TABLE SPACE
110shape_table=HIMEM+1
112amplitude_table=shape_table+steps
114loop_table=amplitude_table+steps
115PROCset_tables
120PROCassemble code
140REM **** BASIC TEST PROGRAM ****
160CLS
170PRINT:INPUT"AMPLITUDE FACTOR 0-7";AF
180IF AF<0 OR AF>7 THEN 170
190?div_factor=AF
200PRINT:INPUT"DELAY FACTOR 1-101";DF
210IF DF<1 OR DF>101 THEN 200
220?delay_factor=DF
230PRINT:INPUT"DURATION FACTOR 0-15";TF
240IF TF<0 OR TF>15 THEN 230
250?time_factor=TF
265REPEAT
270CALL freq
280A$=GET$
290UNTIL A$="X"
300GOTO 160:REM RESTART
900END
999:
1000DEF PROCassemble_code
1005DIM MC% &FF
1010FOR opt%=0 TO 3 STEP 3
1020P%=MC%
1030count=P%:P%=P%+2
1040div_factor=P%:P%=P%+1
1050delay_factor=P%:P%=P%+1
1060time_factor=P%:P%=P%+1
1070[
1075OPT opt%
```

```
1080\**** SET AMPLITUDE TABLE ****
1090\
1095.freq
1100        LDY div_factor
1110        BNE cont
1120        LDA #shape_table MOD 256
1130        STA nexval+1
1140        JMP initc
1150\
1160.cont
1170        LDA # amplitude_table MOD 256
1180        STA nexval+1
1190        LDX #steps
1195.next
1200        LDY div_factor
1210        LDA shape_table,X
1220.more
1230        CLC
1240        ROR A
1250        STA amplitude_table
1260        DEY
1270        BNE more
1280        DEX
1290        BPL next
1300\
1310\**** SET COUNT VALUE ****
1320\
1330.initc
1340        SEI
1350        LDA #0
1360        STA count+1
1370        LDX delay_factor
1380        LDA loop_table,X
1390        LDY time_factor
1400        BEQ nomult
1410.mult
1420        ASL A
1430        ROL count+1
1440        DEY
1450        BNE mult
1460.nomult
1470        STA count
1480\
1490\**** MAIN PROGRAM LOOP ****
1500\
1510.main
1520        LDX #0
1530.nexval
1540        LDA amplitude_table,X
1550        LDY delay_factor
1560.mordel
1570        DEY
1580        BNE mordel
1590\
1600        STA port
1610        INX
1620        CPX #steps
1630        BNE nexval
1640\
1650\**** DECREMENT COUNT ****
1660\
1670        LDA count
1680        SEC
1690        SBC #1
1700        STA count
1710        LDA count+1
1720        SBC #0
1730        STA count+1
1740        BNE main
1750        LDA #0
1760        CMP count
1770        BNE main
1780        CLI
1790        RTS
1800]
1810NEXT opt%
1820ENDPROC
1999:
2000DEF PROCset_tables
2005x=0
2010FOR I=shape_table TO shape_table+steps-1
2020y=127*SIN(x)+127
2030? I=y
2040x=x+2*PI/steps
2050NEXT I
2060:
2070FOR delay=0 TO 101
2080loop_val=10^6/(50*(271+75*delay))
2090loop_table?delay=loop_val
2100NEXT delay
2120ENDPROC
```

CAMERA OBSCURA

The BBC Micro offers users a wide range of graphic modes, enabling complex pictures to be built up onscreen. Numerous graphic ROMs are also available, and now the EVI Video System allows an onscreen image to be produced by simply pointing a camera at the desired object.

The EVI Video system — also known as 'Snap' — is priced to fall well within the budget of a home computer user. Complete with software, it costs only £130 — less than half the price of any comparable video interface. The Snap system consists of a small electronic camera that is connected by a ribbon cable to the BBC's user port, together with software on either cassette or disk. With this system you can transfer the image of any object or scene onto your computer's screen simply by issuing the correct command. You can use Snap just for fun, but more serious applications are possible — the system could form the basis of an 'intelligent' burglar alarm, or could be used for image recognition, perhaps even giving 'sight' to a robot.

Snap works by virtue of a quirk of electronics. Inside the camera, behind the lens, is a 32 Kbyte RAM memory chip. Unlike normal chips, this has been manufactured with a transparent window in the top surface of the chip. The image from the lens is focused onto the surface of the silicon wafer that makes up the chip, on which there is an array of 256 by 128 tiny memory cells. These cells, like those in any other chip, have a property that is usually 'invisible' to the user. When a cell is exposed to light it slowly loses its stored charge, and the rate at which it discharges is proportional to the intensity of the light falling on it. In the Snap camera all the cells are first fully charged by writing a specific value to all memory locations, thus turning each one 'on'. After a short period, any cells receiving light slowly discharge. After a pause, all cells are re-read to ascertain the value in each. Those that have received no light will still have the same 'on' value, while those in the path of sufficient light will have discharged, changing their stored value, and so will be read as 'off'. The system software plots a lit point on the screen in a position that corresponds to each 'off' cell in the memory array. In this way, a reproduction of the image on the chip is transferred to the BBC screen.

The time period between the computer writing all the memory cells to 'on' and then reading them again determines the 'exposure' of the picture. The Snap camera is capable of producing many pictures each second if the lighting is bright enough; in normal room lighting you can take a picture in under a second.

EV1 Moving Pictures
The software with the Snap camera is called EV1. It continuously displays on the screen what the camera 'sees'. It also allows pictures to be dumped to a printer, frozen on screen, or saved to disk. The computer finds what it estimates as an appropriate exposure setting, although this can be changed manually by pressing the up and down arrow keys

The picture resolution is limited by the 256 by 128 array of memory cells on the camera chip. This is not particularly high — it is about the same as a Mode 2 or Mode 5 BBC screen — but it gives a very reasonable picture quality, comparable to a newspaper photograph. A more important restriction on quality is not the resolution but the result of another feature of the memory chip. The array of cells is, in fact, split into two separate segments on the chip surface. A gap between these two blocks means that a strip across the centre of the image is not detected by the camera, and the resultant screen image thus has a small missing section. Surprisingly, this is not too serious and on many pictures is entirely unnoticeable.

The Snap camera itself is encased in a small plastic box about the size of a cigarette pack. At the front is the lens and on the base is a standard

camera tripod mounting, enabling the unit to be steadied when long exposures are needed. Two metres of eight-way ribbon cable connect the unit to the BBC user port. This is all the hardware that is needed, so the system is extremely easy to set up.

The camera uses a lens from the small Pentax 110 single lens reflex camera, fitted in a standard bayonet mounting. Such lenses are readily available from camera shops and are supplied in a wide variety of focal lengths — even zoom lenses may be fitted — so a different lens may be substituted for the one supplied. The only problem with this arrangement is that the lens in the Snap system is set at a different distance from the chip than it would be, on the Pentax camera, from the film. Thus the focusing distance marks on the lens have no meaning when used on the Snap system. Since focusing the Snap camera can be a time-consuming task, it would be worth the effort involved in recalibrating these marks.

The screen image produced by the camera is largely dependent on the software. A suite of programs is supplied with the system, and a 50-page instruction manual (prepared very professionally on an Apple Macintosh) explains software use and the camera's working details.

The camera may be used in two different ways. Pictures may be produced from a single image, resulting in a 'two-tone' screen display, or a 'multi-tone' image may be built up by taking several pictures at different exposures. The first method is used by the first program in the supplied software; this uses a small section of a Mode 4 screen to present a constantly updated picture. The exposure is adjusted by using two of the cursor keys. This program also includes a screen dump routine that is suitable for Epson, or Epson-compatible, printers.

The second program allows more realistic pictures to be constructed in several tones. Eight separate pictures are taken at different exposures, and these are displayed simultaneously on a Mode 0 screen, using shading to give more emphasis to the short exposure images. The effect of this is an image containing eight different grades of brightness, from black through various grey shades to white. This gives a more lifelike effect, but under normal room lighting these 'grey scale' images can take up to 10 seconds to produce — hardly a 'snap'.

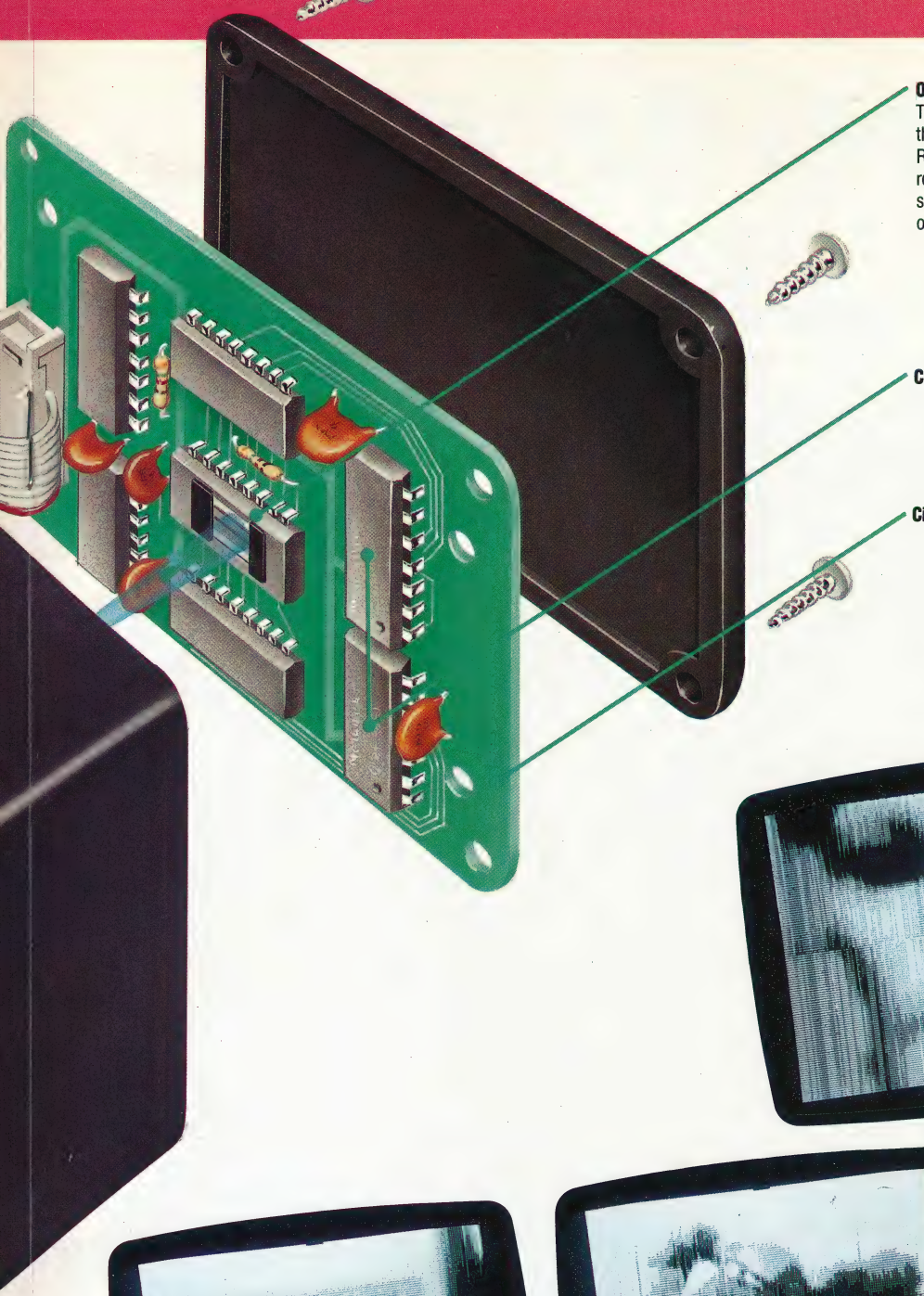
The 'Secure' program turns your snap system into a computerised burglar alarm. The software takes note of only the differences between successive images, so you can set it to give an alarm signal when these differences rise above a certain level. For example, if the camera was pointed at the outside of your house it would ignore trees swaying in the wind but would trigger the alarm if a visitor arrived at the front door.

'Movie' stores a short sequence of two-tone frames, which are then replayed quickly, giving the effect of animation. Also supplied is a program called 'Animal'. This is a video version of the familiar computer game (see page 252). To play

this, you present the system with a picture containing several objects. The program analyses the image, taking note of the outlines of all the separate objects it can 'see' and then invites you to name each one. If you then point the camera at another picture or scene it will attempt to identify any of the objects that also appeared in the first picture.

Perhaps the most attractive option for home users is the program called 'Arty'. This is used to produce complex screen pictures from different images. The whole Mode 1 screen is used, and images captured by the camera can be positioned anywhere on the screen, magnified or reduced in size, by using a joystick. Foreground and background colours can be changed by using the function keys. A degree of patience is needed to make the most of this program, but complicated full-colour screens can be constructed from a variety of real-life objects.

Although the software provided is complex and



SNAP/EV1 VIDEO
Price: £130
Dimensions: 70 mm × 50 mm × 25 mm
Lenses: 24 mm or 18 mm lens
Interfaces: 20-way connector
Manuals: Snap camera user guide
Advantages: Enables users to save pictures either to screen or to disk
Disadvantages: The resolution is poor, which sometimes makes controlling the camera difficult

Vertical Resolution
Pictures generated by the Snap camera are formed entirely from vertical lines, with gaps to show variations in brightness. The resolution of the images is acceptable, but far from outstanding, even considering the £130 price

Lighter Shades Of Pale
Snap camera pictures are normally displayed in two colours, black and white. A grey-scale program called 'Grey' lets you display a picture over the whole screen in eight levels of brightness. The increased ability to show contrasts in shading improves the realism of the displayed image

well thought out, it covers a limited range of applications. The grey-scale picture program, for example, may be used in Mode 0 only. An option to use Mode 2, using the different colours available in that mode as grey tones, would have been more useful. A large amount of information about the machine code routines used by the software is provided in the manual, but to make use of this a reasonable knowledge of machine code is needed. The BASIC-only user is restricted to the software provided. It would have been preferable to have some all-purpose machine code modules that the user could string together to meet any particular requirements. However, this would no doubt have detracted from the most attractive feature of the Snap system — its price.

MAKING PLANS

Despite many basic similarities of design, financial modelling packages each have their own individual characteristics and idiosyncrasies. This is certainly true of Graph Plan, a combined spreadsheet and graphics package, which is available for the BBC Model B equipped with the Z80 second processor.

Unlike the other packages that we have looked at in this series, Graph Plan is a disk-based program. Acorn gives the package away, as part of a bundle of free software, to purchasers of the Z80 second processor. Like all the programs in that bundle, Graph Plan is a very useful and reliable package.

The program doesn't have quite as much 'style' as Psion's Abacus, the spreadsheet/graphics package that Sinclair QL users receive free of charge (see page 724). But it does have the enormous advantage over Abacus of the fast access and storage speeds that proper disk-based software enjoys (the QL Microdrives are adequate if you are used to cassette-based software, but frustratingly slow if you have ever used a proper disk drive). Since Graph Plan is a 'giveaway' package, it is sure to enjoy great popularity — although not as much as Abacus, perhaps, since the combined price of the Z80 processor plus Acorn disk drives is a lot higher than that of the Sinclair QL.

In this article, we will concentrate our attention on the graphics side of financial modelling. Graph Plan, as the name suggests, has a very extensive graphics capability, as well as a formidable array of built-in commercial and mathematical formulae.

What gives Graph Plan its unusual style is its idiosyncratic way of communicating by numbers; all user interaction with the program is via 144 numbered commands. Upon loading, the program has a standard spreadsheet display — divided into rows and columns — filling most of the screen. Down the right-hand side of the screen the 20 basic commands with their corresponding numbers are displayed. The user selects a command and keys in the number at the ENTER COMMAND prompt on the third status line above the display.

Although it is simple enough to select a command — either from the menu on the screen or from the complete list of commands in the excellent 124-page Graph Plan manual — there are obvious disadvantages to this way of doing things. Most modelling packages, particularly the highly successful ones like the Lotus 1-2-3 (see page 644), require you to input the initial letter of a command only (or to select it from a display with the cursor control arrows).

Sophisticated modelling packages, like Lotus 1-2-3, display explanations of what each command's function is. Graph Plan, on the other hand, expects you to understand the function of all of its commands. However, the program does provide the facility for the command list on the right of the screen to be altered to display corresponding lists of commands. If, for example, you select command number 2 data, the command menu changes to display commands 29 to 48 (the data entry and data manipulation commands). There is also a HELP facility (command number 7) that can give an explanation of a given command's function.

In addition to the system of numbered commands, Graph Plan has other unique features. Most spreadsheets, for example, are based around the concept of the 'cell' — an intersection between a row and a column. Graph Plan treats rows and columns as separate entities, and the 'data pointer' (the second status line on the display) is a cursor indicator that, in addition to displaying the identity of the current cursor square, shows whether you are in row or column mode.

This distinction would be meaningless in a system where the unique cell address is the central reference point but it is extremely important in Graph Plan, because graphs have to be drawn up with reference to either rows or columns, but not both. In order to tell the package whether you want to generate a row-based graph or a column-based graph, you have to set the appropriate mode by changing the data pointer. This is done using the arrow keys to move the cursor onto the heading of either a row or a column, which automatically specifies the mode.

A SIMPLE MODEL

As an illustration of the package's graph drawing technique, let's consider a simple model. This has five columns, headed 'January' to 'May' respectively, and five rows, headed 'Sales', 'Cost of Sales', 'Gross Profit' 'Overheads' and 'Net Profit'. In a model like this, a row-based graph will have a different meaning from a column-based graph. For example, we could generate a very simple row-based bar graph for the sales turnover figures for January through to May.

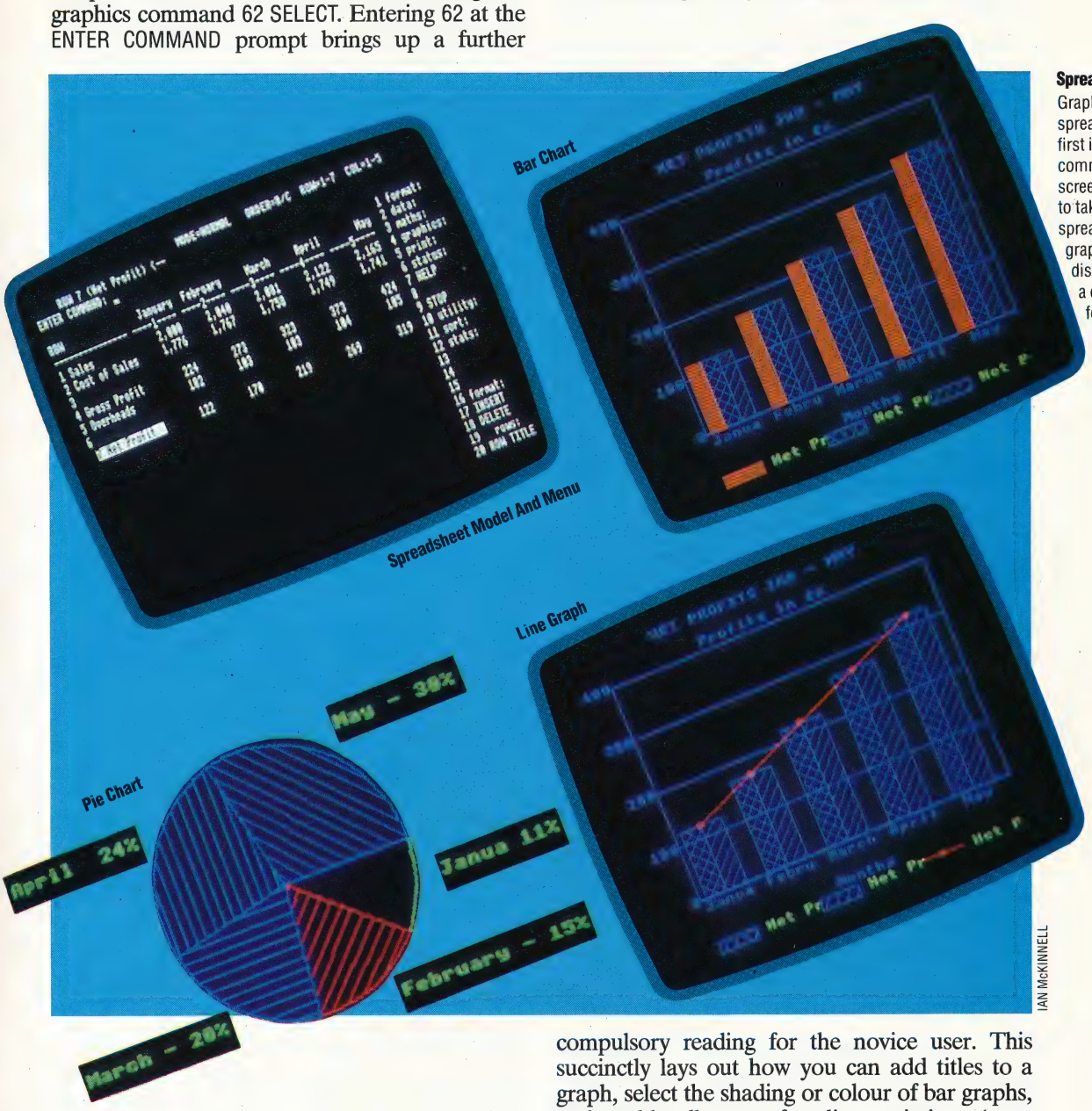
The graph would display the column titles ('January' to 'May') along the x-axis, and the bars would represent the values given in row one of the model. Alternatively, by adjusting the data pointer to column mode, we could generate a very different bar graph. This would display the row

titles ('Sales', 'Cost of Sales', etc.) along the x-axis, with the bars representing the values given in column one ('January') of the model.

Given the appropriate data, Graph Plan can instantly produce a considerable number of different graphs from the data given in this model. All of these graphs can be seen, in turn, on the screen, with no further intervention from the user. Graph Plan allows this to be done using the graphics command 62 SELECT. Entering 62 at the ENTER COMMAND prompt brings up a further prompt on status line three. If you are in 'row mode', for example, you will be asked to give the number of the row of data that you want graphed. As soon as you select a row, the prompt changes to ask which type of graph you want: 'Choose (Bar=1, Line=2, Pie=3)'. You can then view the graph immediately by selecting command 61 DISPLAY. And you can change from bar graphs to line or pie graphs at will.

If you want more flexibility in the layout and design of a graph, you can use command 63 OPTIONS. The manual displays a very good 'decision chart', which clearly illustrates the selection process you have to go through when using this command. Command 63 presents you with a six-option menu screen: Display Chart, Define Chart Options, Define Axes Options, Define Pie Options, Print Chart and Plot Chart. Furthermore, the Graph Plan manual has a special appendix, called a 'Guide through the graphics sub-menus', which is compulsory reading for the novice user. This succinctly lays out how you can add titles to a graph, select the shading or colour of bar graphs, and enable all sorts of scaling variations (even down to making the axes logarithmic rather than linear).

A good graphics facility built into a spreadsheet (that itself has mathematical and statistical functions) makes Graph Plan suitable for a wide range of relatively simple scientific and engineering applications. The package provides an excellent means of presenting data, either for reports or lectures, and will, therefore, appeal to technicians and scientists as well as to commercial users.



IAN McKINNELL

Spreadsheet Display
Graph Plan differs from other spreadsheets in two ways. The first is the menu of numbered commands on the right of the screen. The second is the ability to take data from the spreadsheet and present it in graphic form. Graph Plan can display data from a row or from a column, in any of three formats, as shown

POETRY IN MOTION

In this article in our LOGO series, we turn our attention to list processing, which is central to the way the language works. We also take another look at recursion (and revisit the psychoanalyst) before using LOGO to write a little poetry.

A list is a collection of objects in order, and is identified in LOGO by using square brackets; so that [CEYLON MADRAS VINDALOO] is a list. We have encountered lists several times in this series. In fact, we can't escape from them in LOGO, because the language is based on lists. We've already seen how a definition for a square — REPEAT 4 [FD 50 RT 90] — has a list of instructions (within the square brackets) as its second input. Similarly, MAKE "INP REQUEST assigns to INP a list consisting of the input from the keyboard.

Lists can be assigned to global variables — for example, MAKE "CURRY [CEYLON MADRAS VINDALOO]. The command PRINT :CURRY prints the list without the square brackets: that is, CEYLON MADRAS VINDALOO.

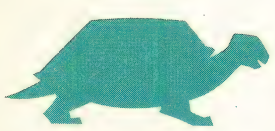
A LOGO object can be a number, a word or a list; and a list is defined as simply a collection of objects. This is, of course, a recursive definition; a list can contain another list, or a list of lists, and so on. [[CHICKEN TIKKA] NAN SALAD] is a valid list, with a list as its first element ([CHICKEN TIKKA]). Recursive procedures are often needed to process lists, precisely because lists are recursive objects.

The majority of our programming in LOGO has until now been concerned with one number or one word at a time. When we want to process groups of objects at the same time, we need to organise these simple objects into a single unit. LOGO takes the list as its basic method of grouping simple objects. The list is chosen because it is extremely versatile — you can construct any complex data organisation by starting from a list.

The two fundamental list operations are FIRST and BUTFIRST. FIRST [CEYLON MADRAS VINDALOO] outputs CEYLON — that is, it gives us the first element of the list. BUTFIRST [CEYLON MADRAS VINDALOO] outputs MADRAS VINDALOO; in other words, it gives us the list without its first element.

Here's a procedure that prints the elements of the list, one below the other:

```
TO PRINTOUT :LIST
   PRINT FIRST:LIST
   PRINTOUT BUTFIRST :LIST
END
```



So PRINTOUT [CEYLON MADRAS VINDALOO] gives:

```
CEYLON
MADRAS
VINDALOO
```

The first command prints the first element of the list and then passes the task of printing the rest of the input list to another copy of PRINTOUT. When you run this procedure you'll get an error message when it runs out of data. Here's a more elegant way of finishing:

```
TO PRINTOUT :LIST
   IF EMPTY? :LIST THEN STOP
   PRINT FIRST:LIST
   PRINTOUT BUTFIRST :LIST
END
```

EMPTY? checks to see if its input is the 'empty list' — []. Some MIT versions do not have the primitive EMPTY?, but you can always define it as follows:

```
TO EMPTY? :LIST
   IF :LIST = [] THEN OUTPUT "TRUE
   OUTPUT "FALSE
END
```

Similar to FIRST and BUTFIRST are LAST and BUTLAST. LAST [CEYLON MADRAS VINDALOO] outputs VINDALOO, and BUTLAST [CEYLON MADRAS VINDALOO] outputs CEYLON MADRAS.

BABBLING

For our first exploration in list processing, we'll try to mimic some random babblings on the psychoanalyst's couch. First we'll assign all the words we know to the variable WORDS:

```
MAKE "WORDS [MOTHER FATHER SEX MURDER
JEALOUSY FIRE SEA DEATH DREAM]
```

We want to produce a constant random stream of these words, for experience has shown us that these are the words that are always successful in attracting our psychoanalyst's attention. To get a random element of the list we need to select a random number, n, between one and the length of the list (nine in this case) and then select the nth element of the list.

```
TO NTH :NO :LIST
   IF :N = 1 THEN OUTPUT FIRST :LIST
   OUTPUT NTH :NO - 1 BUTFIRST :LIST
END
```

Let's use this procedure with a few examples to see how it works. Say you type NTH 1 :WORDS. The condition in the first line is true, so the procedure outputs FIRST :WORDS, which in our example is MOTHER.

Try NTH 2 :WORDS — now the condition is false

so the procedure outputs NTH 1 BUTFIRST :WORDS. This ignores the first list element and takes the first word from the remainder of the list — FATHER.

So our procedure to print a random word from our limited vocabulary would be:

```
TO GETRANDOM :LIST
  OUTPUT NTH ( ( RANDOM 9 ) + 1 ) :LIST
END
```

To use this, type GETRANDOM :WORDS.

Our procedure is restricted to lists of nine items. We could improve on this if we could determine how many items there are in a given list. Here is a procedure that does this:

```
TO LENGTH :LIST
  IF EMPTY? :LIST THEN OUTPUT 0
  OUTPUT 1 + LENGTH BUTFIRST :LIST
END
```

To see how this works try: LENGTH [SCIENCE FICTION]. As the list contains some words the first condition fails, so the procedure outputs 1 + LENGTH [FICTION]. Now LENGTH [FICTION] outputs 1 + LENGTH []. On calling LENGTH with an input of [], the condition in line 1 is true, so the procedure outputs 0. Now LENGTH [FICTION] outputs 0 + 1 = 1 and, finally, LENGTH [SCIENCE FICTION] outputs 1 + 1 = 2. So a more general procedure for getting random words from a list of any length is:

```
TO GETRANDOM :LIST
  OUTPUT NTH ( ( RANDOM LENGTH :LIST ) + 1 )
  :LIST
END
```

In many versions of LOGO there is a primitive, ITEM, which does precisely what NTH does, and a primitive called COUNT that does the same as LENGTH. Using these we can rewrite the procedure:

```
TO GETRANDOM :LIST
  OUTPUT ITEM ( ( RANDOM COUNT :LIST ) + 1 )
  :LIST
END
```

To print a selection of 10 comments to keep your psychoanalyst listening attentively, simply type:

```
REPEAT 10 [PRINT GETRANDOM :WORDS]
```

There is a pattern to these list processing programs that was shared by many of our recursive turtle graphics procedures. The pattern is:

- If the task to be performed is extremely simple then do it and stop.
- Otherwise do a small part of the task.
- Then pass the rest of the task onto another procedure (often a copy of the original procedure).

This is a highly successful strategy, which we will encounter repeatedly in list processing programs. Compare this polygon drawing program:

```
TO POLY :N
  IF :N = 0 THEN STOP
  FD 30 RT ( 360 / :N )
  POLY :N — 1
END
```

with the version of PRINTOUT given earlier. The structure of the two procedures is identical.

RANDOM POETRY

Having failed to impress our psychoanalyst, we now turn our hand to poetry. Here, we will want to produce whole sentences rather than single words.

```
TO POEM1 :LENGTH
  IF :LENGTH = 0 THEN PRINT "STOP
  ( PRINT1 "' ' GETRANDOM :WORDS )
  POEM1 :LENGTH – 1
END
```

PRINT1 "' ' is included to print a space between words. To use this procedure, type POEM1 6 for a six-word sentence.

It would be useful to be able to extend our

Abbreviations

BUTFIRST	BF
BUTLAST	BL
SENTENCE	SE

SIR JOHN TENNIEL

Mock Turtle Sings
An extract from the Mock Turtle's song from 'Alice in Wonderland' by Lewis Carroll. The metric pattern, a little difficult to duplicate in a computer poem, is adapted from an old folk song, and is also utilised by Mary Howitt in the classic poem, 'The Spider and the Fly'

'Will you walk a little faster?' said a whiting to a snail,
'There's a porpoise close behind us, and he's treading
on my tail.
See how eagerly the lobsters and the turtles all
advance!
They are waiting on the shingle — will you come and
join the dance?
Will you, won't you, will you, won't you,
will you join the dance?
Will you, won't you, will you, won't you,
won't you join the dance?'

'You can really have no notion how delightful it will be
When they take us up and throw us, to the lobsters, out
to sea!'
But the snail replied 'Too far, too far! and gave a look
askance —
Said he thanked the whiting kindly, but he would not
join the dance.
Would not, could not, would not, could not,
would not join the dance.
Would not, could not, would not, could not,
could not join the dance.

original list of words without having to go to the trouble of writing it all out again. One way of extending a list is to use the operation SENTENCE, which takes two inputs and makes a list from them. So SENTENCE "JAM [HONEY JAR] outputs [JAM HONEY JAR].

```
TO ADDWORDS1 :LIST
   MAKE "WORDS SENTENCE :LIST :WORDS
END
```

So we can now extend WORDS with ADDWORDS [ANXIETY REPRESSION [FEAR OF FLYING]]. The problem with this is if the variable WORDS has not previously been assigned a value. The primitive THING? is used to overcome this by testing if a variable has been assigned a value; it outputs true if its input has a value associated with it. We can now improve our list of extra words with ADDWORDS1:

```
TO ADDWORDS1 :LIST
   IF NOT THING? "WORDS THEN MAKE "WORDS []
   MAKE "WORDS SENTENCE :LIST :WORDS
END
```

Using a different list of words, we obtained the following piece of 'poetry' using this procedure:

APPARITION LOUDLY SPOKE SPLENDID PARANOID PLANET TERRIFIED THE WITH GREEN APPARITION FLOATING PARANOID ROBOT MAN FLEW SPOKE FLOATING LOUDLY

One of the more obvious failings of our computerised poetry is its total disregard for English grammar. The poems might make more sense if we could constrain them to some simple syntactical patterns — such as: noun, verb, noun. One way to do this is to have a number of lists, one for each part of speech. We could then choose one word from each list according to our desired sentence structure.

We leave this problem for you to explore and investigate. In the next instalment of the course, we will show you some ways of how to improve the turtle's poetry-writing abilities.

Logo Flavours

Some versions of MIT LOGO do not have EMPTY?, ITEM and COUNT. In all LCSI versions, use:

EMPTYP for EMPTY?
LISTP for LIST?
TYPE for PRINT1

There is a primitive, EQUALP, which tests whether its two inputs are the same. Use this for comparing lists and words in place of the equals sign (=). (The equals sign works for lists on some LCSI versions, but not on others.)

Remember the different IF syntax:
IF EMPTYP :LIST [OUTPUT 0]
On Atari LOGO use SE for SENTENCE, and note that ITEM is not implemented

Exercises

1) Write a procedure to print a list in reverse order (use LAST and BUTLAST). Modify this procedure so that it outputs the reversed list
2) Write a procedure that removes an element from a list. So DELETE "FOOD [DRINK FOOD] outputs [DRINK] and DELETE "WINE [DRINK FOOD] outputs [DRINK FOOD]

Exercise Answers

Answers to the exercises on page 737:

1. Calculation powers:

```
TO POWER :A :N
   IF NOT ( ( INTEGER :N ) = :N ) THEN PRINT
   [WHOLE NUMBER INDICES ONLY] STOP
   IF :N = 0 THEN OUTPUT 1
   OUTPUT :A * POWER :A :N - 1
END
```

2. Converting to hexadecimal:

```
TO HEX.PRINT :NO
   IF :NO < 10 THEN OUTPUT :NO
   IF :NO = 10 THEN OUTPUT "A
   IF :NO = 11 THEN OUTPUT "B
   IF :NO = 12 THEN OUTPUT "C
   IF :NO = 13 THEN OUTPUT "D
   IF :NO = 14 THEN OUTPUT "E
   IF :NO = 15 THEN OUTPUT "F
END

TO HEX :NO
   IF :NO = 0 THEN STOP
   HEX QUOTIENT :NO 16
   PRINT1 HEX.PRINT REMAINDER :NO 16
END
```

3. Testing if a number is even:

```
TO EVEN? :NO
   IF ( ( REMAINDER :NO 2 ) = 0 ) THEN OUTPUT
   "TRUE OUTPUT "FALSE
END
```

4. Finding an area using the Monte Carlo method:

```
TO MC
   DRAW PU MAKE "IN 0
   MC1 1000 10 100
   ( PRINT [AREA IS] ( :IN ) )
END

TO MC1 :NO :XNO :YNO
   IF :NO = 0 THEN STOP
   RANDOM.POINT :XNO :YNO
   IF INSIDE? THEN MAKE "IN :IN + 1
   MC1 :NO - 1 :XNO :YNO
END

TO RANDOM.POINT :XNO :YNO
   SETXY RANDOM :XNO RANDOM :YNO
END

TO INSIDE?
   IF YCOR < XCOR * XCOR THEN OUTPUT "TRUE
   OUTPUT "FALSE
END
```

H

HERTZ

Named after scientist Heinrich Hertz (1857-94) the *hertz* is a unit of measurement for frequency. When a repetitive event recurs once every second, it has a frequency of one hertz. This measurement is applied to sound generation, as in the frequency of a tone; in electronics, when an electrical pulse is repeated; and in video scanning, where the hertz value refers to the frequency with which lines of resolution are traced by a beam of light.

The abbreviation for hertz is Hz; *kilohertz*, or kHz, which stands for thousands of cycles per second; and *megahertz*, or MHz, for millions of cycles per second.

HEURISTIC

A *heuristic* system is one that relies on a self-learning, or trial and error, approach to problem-solving. This is only one of two basic ways of solving problems. The other method involves creating a system that applies a specific set of rules or instructions, using existing knowledge of the nature of the problem. For example, solving simple equations for an unknown value is accomplished by following a specific set of mathematical operations on the equation. The set of instructions or operations needed to solve the problem is called an *algorithm*. When an algorithm is known, or can be determined by synthesising known data into an original approach, the system is said to be 'non-heuristic'.

A heuristic system, on the other hand, is one that follows a specific course of action up to a point, but then proceeds to 'learn' the best way of finding a solution. This is usually done through trial and error, via some form of feedback system. Heuristic techniques are employed when a decision is required for which the computer has no clear-cut procedure to enable it to make a choice. One example of a heuristic system is the micromouse (see page 721), designed to explore and solve a maze that it has never experienced before. The programmer of a micromouse can give it some very specific instructions, but the mouse must rely on feedback from its sensors to 'learn' the structure of the maze.

Algorithms can be applied to the search at many points along the way. For instance, the mouse has an internal definition for a blank wall, and a set of instructions telling it what to do when one is found — but the mouse must rely on information from its sensors to know that a wall has been found. By trial and error, the mouse finds its way to the end of the maze, learning the optimum path along the way.

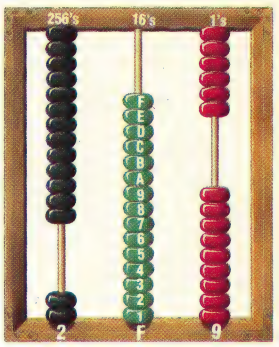
Self-learning, or heuristic, methods of problem-solving are crucial in human intellectual development. In computers, they form a cornerstone of artificial intelligence.

HEXADECIMAL

Hexadecimal notation is a system of representing numerical values using base 16 — as opposed to binary numbers, which are base 2, and decimal numbers, which are base 10. In hexadecimal notation, the digits 0 to 9 are followed by the letters A to F, so that values from 0 to 15 can be represented by a single digit.

Hexadecimal notation is widely used in Assembly language code because it requires much less space than the equivalent binary value. For example, the value 255 requires three digits in decimal notation. Its binary equivalent, 11111111, requires eight digits, but the hexadecimal form, FF, requires only two.

Some microcomputer systems incorporate a *hex-pad* to simplify entering hexadecimal numbers. The pad consists of 16 keys, which are labelled 0 to 9 and A to F.



MARK WATKINSON

Hexadecimal Abacus
To calculate in hexadecimal values would require an abacus with 15 beads per rod. The beads stand for decimal values one through 15, represented in hex notation by the digits 0 through 9 and letters A to F. The rightmost rod represents units, then proceeds to the left in multiples of 16. Thus, each bead on the second rod from the right equals 16, the third 16^2, or 256, and so on. The number shown is decimal 761 (reading from the left, $2 \times 256 + 15 \times 16 + 1 \times 9$). This is read as '2F9 hex'

HIERARCHICAL COMMUNICATIONS SYSTEM

A communications network divided into levels of responsibility is called a *hierarchical communications system*. The lowest levels of the network have the most specific function, while each successive level above has a more general responsibility, and relies on the information processed at the levels below it. Perhaps the best way to examine this is to consider how a communications systems works within the operational structure of a large corporation, with offices in several locations.

Each regional office has a department whose primary purpose is to process incoming orders from locations within its area. Each order department will have its own local area network to process the data, and the department manager will be in charge of the central supervising system for that department. This supervising 'node' is then connected, via a longer-distance network, with the controlling nodes of the other order departments. The company's order manager at head office then has immediate access to data from all the separate regions. This system is, in turn, in communication with those of other central office managers.

In a *hierarchical file system*, there are several levels of files, each lower level dependent on, and related to, the next higher level. A document is stored in a file, which is a subheading of a larger file, and so on. In this system a point is reached where a single umbrella file stores all the units.

HI-RES GRAPHICS

Hi-res, short for high resolution, simply means that a graphic image is built from a large number of very small dots (pixels). The higher the resolution, the larger the number of pixels used in a graphic display — which gives a more detailed, crisper image. In practice, hi-res graphics vary according to a machine's highest level of resolution. Some computer graphics systems are capable of producing a resolution of several thousand lines. Most microcomputer manufacturers use the term to indicate the highest level of resolution their machine can produce.

BUG REPELLENT

To illustrate the techniques of the top-down design approach to Assembly language programming, we now begin to build up a debugging program. The first thing we must do is develop the control module, which has overall command of the lower-level modules that perform more specific activities.

We shall first take a brief look at the specification and design stages for the production of a debugger. The specification is reasonably straightforward; we have already looked at the functions we would expect such a program to provide (see page 739).

The inputs to the debugger will be:

1. A program to debug: We will assume that the debugger is loaded with the program it is to debug already in memory.
2. Commands: We must decide whether the commands are to be entered directly or as choices from a menu. We will enter single-character commands from the list given in the margin.
3. Addresses: These would presumably be entered in hex, so it will be necessary to convert a string of ASCII hex digits to a 16-bit binary number.

The outputs from the debugger will be:

1. 'Echoes' of input characters: Remember that keypresses do not automatically generate characters on the screen — the computer must be programmed to do this (this is called 'echoing').
2. Eight- and 16-bit numbers: These are accepted as strings of hex digits.
3. Strings: These are used to label the above.

There are many ways in which a program could be split up into modules and then into subroutines, but there must always be an outer module — the 'shell' — which ties all the others together. For our debugger program, this will take the form:

B	insert Breakpoint
U	Un-insert (remove) breakpoint
D	Display current breakpoints
S	Start running program
G	Go (resume from where the program left off)
R	display contents of Registers
M	inspect and change Memory location
Q	Quit

THE MAIN MODULE

Data:
- **Start-Address** of program (16-bit)
- **Prompt** for command entry (single ASCII character '>')
- **Command Character** is a single ASCII character (do we allow lower-case characters?)
- **Break-Address** is the address of the handler routine that services the SWI interrupt

Process:

```
Set up Interrupt
GET Start-Address
REPEAT
    DISPLAY Prompt
    REPEAT
        Get Command
    UNTIL Command is valid
    DISPLAY (Echo) Command
    IF Command = 'B' THEN
        Insert-Breakpoint
    ELSE IF Command = 'U' THEN
        Remove-Breakpoint
    ELSE IF...
  Until Command = 'Q'
```

End of Main Module

From this we now have a good idea of the routines that will be required. A module is not the same thing as a subroutine, however. Clearly, there are several subroutines that logically go together in groups with shared data — one such module, for example, might deal with breakpoints. The next stage of refinement shows how we might design such a module:

MODULE BREAKPOINTS

Data:
- **Breakpoint-Table** is an array of 16-bit addresses where breakpoint addresses can be stored
- **Removed-Values** is an array of eight-bit values corresponding to the above table. The op-codes that get replaced by an SWI instruction at the breakpoint can be stored in this
- **Number-Of-Breakpoints** is an eight-bit value containing the number of active breakpoints is an eight-bit value, which contains the next breakpoint that will be encountered in the run
- **SWI-Opcode** is an eight-bit op-code for the SWI instruction

Process1: Insert-Breakpoint

```
IF Number-Of-Breakpoints < MAX THEN
    Get-Address
    Add 1 to Number-Of-Breakpoints
    Store Address in Breakpoint-Table
    (Number-Of-Breakpoints)
ENDIF
```

End Of Process1

Process2: Set-Up-Breakpoint(N)

```
(N tells us which of the breakpoints in the table is to
    be set up)
Get-Address in Breakpoint-Table(N)
Get Op-code at that Address
Store it in Removed-Values(N)
Store SWI-Opcode at Address
```

End of Process2

Process2 is at the stage where we could begin coding it. There are four data values that must be manipulated: N, the parameter that tells us which breakpoint to use, is an eight-bit number in the

range (one to Number-Of-Breakpoints — 1) which we use as an offset into the two tables. Note, however, that one table is of 16-bit values whereas the other is of eight-bit values. We will assume that N is passed in A. The address of the breakpoint obtained from that table will be put into X. The removed op-code will be put into B for transfer to the Removed-Values table. B can then be used to put the SWI op-code into the appropriate address.

We give the final coded form of Process2 (Set-Up-Breakpoint Module) here; our next task is to develop a module to handle input and output. As you will have seen from the design of the debugger so far, there are a number of I/O tasks to be performed by the program. For the moment, we will assume the existence of two subroutines: INCH, which will input a single character into the A register from the keyboard; and OUTCH, which will send a character from A to the screen at the current cursor position. The routines required by this module are:

1. GetCommand: Input the next command from the keyboard.
2. GetAddress: Get a hex address (one to four characters long) from the keyboard.
3. GetValue: Get a hex value (one or two characters long) to modify the value of a memory location.
4. DisplayValue: Display a two-character hex value on the screen.
5. DisplayAddress: Display a four-character hex address on the screen.

Our approach illustrates the difference between the top-down and the bottom-up methods of programming. The top-down approach might lead us to define and code these operations independently, thus ending up with a number of separate routines that do essentially the same thing. The bottom-up approach can produce a saving in time, effort and space by simply writing a few useful routines that are used in a number of different circumstances. These routines are:

GETCH: To input a single character into A, checking against a list of valid characters (command letters or hex digits), echoing valid characters and ignoring others.
GETHX2: To use GETCH to get two hex digits and convert them into an eight-bit number.
GETHX4: To get four hex digits to form a 16-bit number.
PUTHEX: To display an eight-bit number as two hex digits. (This can be called twice to display a 16-bit number.)
PUTCR: To output a carriage return (or carriage return and line feed if necessary).

These five routines need to be developed in turn. First, we will consider the design of GETCH.

GET CHARACTER ROUTINE

Data:
Inchar is an ASCII character input from the keyboard (held in A)
Valid-Chars holds the 16-bit address of the table of valid characters
Number-Of-Valid-Chars is an eight-bit value
Chars-Searched is an eight-bit counter

Process:
REPEAT
  Get next Inchar
  Set Chars-Searched to (Number-Of-Valid-Chars – 1)
  While Valid-Chars(Chars-Searched) <> Inchar
    AND Chars-Searched >=0
    Decrement Chars-Searched
Until Chars-Searched >=0
DISPLAY Inchar

In order to code this, we must use A to store Inchar, and the 16-bit Valid-Chars value can be passed and kept in X. The Number-Of-Valid-Chars can be passed in B, but will need to be kept more permanently, by pushing it onto the stack. B can then be used for Chars-Searched. Note that B will return the offset into the table, which will be useful in command interpretation and hex conversion.

We give the final coded form of this routine here. In the next instalment of the course, we will develop the other routines required by the input/output module.

GETCH Routine

```
GETCH   PSHS    B           Save B
REPTOO  BSR     INCH        Get Next Inchar
        LDB     1,S         Set Chars-Searched
        DECB                Subtract one from max offset
WHILOO  BLT     ENDWOO      While Chars-Searched >=0
        CMPA    B,X         AND
        BEQ     ENDWOO      Inchar <> Valid-Chars(Chars-
                            Searched)
        DECB                Decrement Chars-Searched
        BRA     WHILOO
ENDWOO  TSTB
UNTLOO  BLT     REPTOO      Until Chars-Searched >=0
        BSR     OUTCH       Display Inchar
        LEAS    1,S         Increment S to 'forget' the
        RTS                 original value of B
```

Set-Up-Breakpoint Module

```
                            Data declarations
BPTAB   RMB     32          Breakpoint-Table
REMTAB  RMB     16          Removed-Values
NUMBP   FCB     0           Number-Of-Breakpoints
NEXTBP  FCB     0           Next-Breakpoint
SWIOP   FCB     $3F         SWI-Opcode
MAXBP   FCB     16          Maximum number of Breakpoints
                            Process2 — Set-Up-Breakpoint
BP02    PSHS    B,X         Save the registers we will alter
        LSLA                Multiply the offset by two
        LEAX    BPTAB,PCR   Base address of table
        LDX     A,X         Get Address in Breakpoint-Table(N)
        LDB     ,X          Get Op-code at that address
        LSRA                Restore A to original value
        STB     A,X         Store Op-code in Removed-Values(N)
        LDB     SWIOP,PCR   Get SWI-Opcode
        STB     ,X          Store it at the address
        PULS    B,X,PC      Restore and return
                            End of Process2
```

RED ALERT

A new American computer game offers Commodore 64 owners the chance to engage enemy fighters in mid-air dogfights. Flight simulation programs are common enough, but Tymac's Flyerfox has the added bonus of built-in speech routines that may be used without any additional hardware.

Although computer games have come a long way since the early days of Space Invaders (see page 615), most of these advances have been in the development of graphics. Programmers have been chiefly concerned with finding new ways of squeezing the data for an increasing number of graphic screens into a limited amount of RAM. In the meantime, the excellent sound capabilities of many home computers have largely been ignored.

Now the American company Tymac has begun UK distribution of a series of games that incorporate speech synthesis without actually using a speech interface. The first of these programs to become widely available is Flyerfox for the Commodore 64. This is a flight simulation program in which the player is 'flying' a fighter plane that has to escort a jumbo jet with a high-ranking government official aboard through disputed airspace. Enemy MiG fighters attempt to shoot down the jet, and the player's objective is to engage and destroy the enemy aircraft. The speech synthesis used in the game involves a series of messages transmitted from the jumbo to the player/pilot.

The speech synthesiser is a part of software that takes up around 11 Kbytes of memory to store the data used to recreate the required phrases. Flyerfox uses the 'linear predictive' coding method. In this system, words are converted into digital signals, which are then stored in RAM. When a particular word is needed, the corresponding digital data is accessed, and the word reproduced via the Commodore's SID chip.

The game's graphics are in high resolution throughout. The screen display consists of the view forwards, showing the sky as seen through the cockpit window and the instrument panel. Various navigation aids are supplied, including a compass and a radar panel that shows the approaching MiG fighters, giving the player time to prepare for combat. A further aid is provided by two flashing lights, one on each side of the artificial horizon, which tell the player if the MiGs are above or below cockpit level.

The dogfight sequences are fast and very realistic. When a MiG appears onscreen, the program produces a warning beep and the player must then manoeuvre the Flyerfox so that the attacker is in the cross-hairs of his gunsight. This is not easy, as the planes dodge and dive at great speed. Once the target is fixed in the sights, the player may then fire the heat-seeking missiles; however, these are not infallible and the enemy fighters often escape.

Although the graphics are of a high quality, they do not offer much variety. The illusion of movement is achieved by changing cloud patterns, and the ground is simply a scrolling grid. It must also be said that the airliner itself adds very little to the game. Leaving aside the obvious parallel with the Korean Jumbo 007, which leaves the game open to charges of tastelessness, it is difficult to see why the airliner is included. It can be viewed from the rear only, and the Flyerfox is unable to overtake it when trying to engage the enemy fighters. Furthermore, unlike a real aircraft, the Jumbo does not even attempt to take evasive action when attacked.

Flyerfox certainly represents a new trend in computer games. Speech synthesis within a program is a subject that has been considered for some time. Now Tymac has produced a game that uses speech but which requires no special interface or hardware device. As such, it may well come to be regarded as a landmark in the development of the computer game.

Flyerfox: For Commodore 64, disk £14.95, cassette £9.95
Publishers: Tymac Corporation
Authors: Gregory Carbonaro, Charles Teufert, Ronald Pintus, Arthur Aspromatis
Joystick: Required
Format: Disk or cassette

Seek And Destroy
The pilot of the Flyerfox can glean as much information about the positions of the enemy fighters from the instrument panel as by searching the sky. The dots on the radar screen are aircraft, although not all will attack. The relative height of the MiGs is shown by the two white squares on either side of the altimeter. The compass aids in relocating the Jumbo

DATABASE

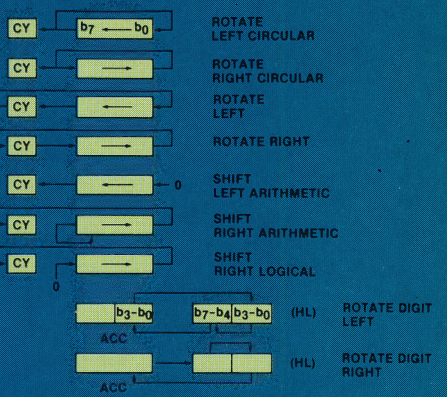
Here, courtesy of Zilog Inc, we reproduce a further part of the Z80 programmers' reference card

Rotate and Shift Group

SOURCE AND DESTINATION

TYPE OF ROTATE OR SHIFT	A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)
'RLC'	CB 07	CB 00	CB 01	CB 02	CB 03	CB 04	CB 05	CB 06	DD CB d 06	FD CB d 06
'RRC'	CB 0F	CB 08	CB 09	CB 0A	CB 0B	CB 0C	CB 0D	CB 0E	DD CB d 0E	FD CB d 0E
'RL'	CB 17	CB 10	CB 11	CB 12	CB 13	CB 14	CB 15	CB 16	DD CB d 16	FD CB d 16
'RR'	CB 1F	CB 18	CB 19	CB 1A	CB 1B	CB 1C	CB 1D	CB 1E	DD CB d 1E	FD CB d 1E
'SLA'	CB 27	CB 20	CB 21	CB 22	CB 23	CB 24	CB 25	CB 26	DD CB d 26	FD CB d 26
'SRA'	CB 2F	CB 28	CB 29	CB 2A	CB 2B	CB 2C	CB 2D	CB 2E	DD CB d 2E	FD CB d 2E
'SRL'	CB 3F	CB 38	CB 39	CB 3A	CB 3B	CB 3C	CB 3D	CB 3E	DD CB d 3E	FD CB d 3E
'RLD'								ED 6F		
'RRD'								ED 67		

	A
'RLCA'	07
'RRCA'	0F
'RLA'	17
'RRA'	1F



Mnemonic	Symbolic Operation	S	Z	H	P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
RLCA		•	•	X	0	X	•	0	I	00 000 111	07	1	1	4	Rotate left circular accumulator
RLA		•	•	X	0	X	•	0	I	00 010 111	17	1	1	4	Rotate left accumulator
RRCA		•	•	X	0	X	•	C	I	00 001 111	0F	1	1	4	Rotate right circular accumulator
RRA		•	•	X	0	X	•	0	I	00 011 111	1F	1	1	4	Rotate right accumulator
RLC r		I	I	X	0	X	P	0	I	11 001 011 00 000 r	CB	2	2	8	Rotate left circular register r
RLC (HL)		I	I	X	0	X	P	0	I	11 001 011 00 000 110	CB	2	4	15	r Reg 000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC (IX + d)	r,(HL),(IX + d),(IY + d)	I	I	X	0	X	P	0	I	11 011 101 11 001 011 ← d → 00 000 110	DD CB	4	6	23	
RLC (IY + d)		I	I	X	0	X	P	0	I	11 111 101 11 001 011 ← d → 00 000 110	FD CB	4	6	23	
RL m	m = r,(HL),(IX + d),(IY + d)	I	I	X	0	X	P	0	I	010					Instruction format and states are as shown for RLC s To form new opcode replace 000 of RLC s with shown code
RRC m	m = r,(HL),(IX + d),(IY + d)	I	I	X	0	X	P	0	I	001					
RR m	m = r,(HL),(IX + d),(IY + d)	I	I	X	0	X	P	0	I	011					
SLA m	m = r,(HL),(IX + d),(IY + d)	I	I	X	0	X	P	0	I	100					
SRA m	m = r,(HL),(IX + d),(IY + d)	I	I	X	0	X	P	0	I	101					
SRL m	m = r,(HL),(IX + d),(IY + d)	I	I	X	0	X	P	0	I	111					
RLD		I	I	X	0	X	P	0	•	11 101 101 01 101 111	ED 6F	2	5	18	Rotate digit left and right between the accumulator and location (HL)
RRD		I	I	X	0	X	P	0	•	11 101 101 01 100 111	ED 67	2	5	18	The content of the upper half of the accumulator is unaffected.

NOTES: e represents the extension in the relative addressing mode.
e is a signed two's complement number in the range <-126, 129>
e-2 in the opcode provides an effective address of pc + e as PC is incremented by 2 prior to the addition of e

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
I = flag is affected according to the result of the operation.